Advances in Database Technology — EDBT 2017

20th International Conference
on Extending Database Technology
Venice, Italy, March 21–24, 2017
Proceedings

Editors

Volker Markl
Salvatore Orlando
Bernhard Mitschang
Periklis Andritsos
Kai-Uwe Sattler
Sebastian Breß

open
proceedings

Advances in Database Technology — EDBT 2017
Proceedings of the 20th International Conference
on Extending Database Technology
Venice, Italy, March 21–24, 2017

Series ISSN: 2367-2005

Editors

Volker Markl, Technische Universität Berlin (TU Berlin), Germany
Salvatore Orlando, Ca' Foscari University of Venice (CFU), Italy
Bernhard Mitschang, Universität Stuttgart, Germany
Periklis Andritsos, University of Toronto, Canada
Kai-Uwe Sattler, TU Ilmenau, Germany
Sebastian Breß, German Research Center for Artificial Intelligence (DFKI), Germany

open proceedings

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

OpenProceedings ISBN: 978-3-89318-073-8 DOI of this front matter: `10.5441/002/edbt.2017.01`

Foreword

The International Conference on Extending Database Technology (EDBT) is a leading international forum for database researchers, practitioners, developers, and users to discuss cutting-edge ideas, and to exchange techniques, tools, and experiences related to data management. Data management is an essential enabling technology for scientific, engineering, business, and social communities. Data management technology is driven by the requirements of applications across many scientific and business communities, and runs on diverse technical platforms associated with the web, enterprises, clouds and mobile devices. The database community has a continuing tradition of contributing with models, algorithms, and architectures, to the set of tools and applications enabling day-to-day functioning of our societies. Faced with the broad challenges of today's applications, data management technology constantly broadens its reach, exploiting new hardware and software to achieve innovative results.

EDBT 2017 solicited submissions of original research contributions, as well as descriptions of industrial and application achievements, and proposals for tutorials and software demonstrations. We encouraged submissions of research papers related to all aspects of data management defined broadly, and particularly encouraged work on topics of emerging interest in the research and development communities.

In addition to regular research paper submissions, EDBT 2017 solicited the submission of research papers that come within special topics of interest: "Vision", "Experiments and Analyses" and "Database Technology and Behavior, Security, Ethics, Rights and Duties of Citizens". These papers were reviewed by the same program committee as regular research papers. However, a dedicated co-chair for each special topic provided specific instructions to the reviewers of these papers and coordinated discussions, decisions, and meta-review formulation.

One innovation of EDBT 2017 is the solicitation of short papers, which are presented as posters at the plenary poster session of the conference. These short papers provide an opportunity to describe significant work in progress or research that is best communicated in an interactive or graphical format. In particular, these works contain smaller or more speculative ideas, controversial research topics, and new applications of old ideas or the reworking of previous studies. Short papers were reviewed by the research program committee in a second, independent call after the regular research paper submissions had been reviewed and decided.

The program committees of EDBT accepted 37 out of 168 submitted regular research papers, resulting in an acceptance rate of 22% for the research track; 22 out of 93 submitted short papers, resulting in an acceptance rate of 23.6% for short research papers; 18 out of 45 demos, resulting in an acceptance rate of 40% for the demonstration track; 9 out of 24 industrial and application papers, resulting in an acceptance rate of 37.5%, as well as 3 out of 8 tutorials, again resulting in an acceptance rate of 37.5%.

The papers will be presented in nine research paper sessions, four industrial and application sessions (one invited), two plenary poster sessions, and two demo sessions. In addition, the program features six workshops, one of which is dedicated to European Research projects with a focus on the Horizon 2020 program, four joint keynotes with the ICDT conference, three tutorials, and one panel on the special topic "Database Technology and Behavior, Security, Ethics, Rights and Duties of Citizens". I would like to thank all authors for their contributions, as a successful conference crucially relies on high-quality submissions. The submission numbers indicate a healthy EDBT community. I also would like to thank all co-chairs and reviewers for serving on the EDBT program committee, in particular for the timely handling of all reviews and discussions with a high degree of professionalism and very high review and discussion quality. This enabled us to notify authors with no or only very little delay despite several reviewing cycles and only one month of reviewing and discussion time. Even though these community service contributions require a lot of work on a tight schedule, they are what make our research community function and ensure the overall impact of research in our field.

I firmly believe that we can look forward to an interesting program and exciting conference on March 21–24, 2017, in Venice.

Volker Markl
EDBT 2017 Program Chair

Program Committee Members

Research Program Committee

Ziawasch Abedjan, TU Berlin
Bernd Amann, U Pierre et Marie Curie
Walid G. Aref, Purdue U
Khalid Belhajjame, U Paris-Dauphine
Michael Böhlen, U Zürich
Klemens Böhm, Karlsruhe Inst. Techn. (KIT)
Francesco Bonchi, Yahoo! Labs
Angela Bonifati, Lille 1 U
Philippe Bonnet, IT U Copenhagen
Sebastian Breß, DFKI
Nieves Brisaboa, U La Coruna
Lei Chen, Hong Kong U of Sc. & Techn.
Sara Cohen, Hebrew U Jerusalem
Dario Colazzo, U Paris-Dauphine
Alfredo Cuzzocrea, U Trieste
Antonios Deligiannakis, TU Crete
Elena Ferrari, U Insubria
Peter M. Fischer, U Freiburg
Avigdor Gal, Technion
Helena Galhardas, U Lisbon
Johann Gamper, Free U Bozen-Bolzano
Minos Garofalakis, TU Crete
Tingjian Ge, U Mass. Lowell
François Goasdoué, U Rennes 1
Goetz Graefe, HP Labs
Michael Grossniklaus, U Konstanz
Moerkotte Guido, U Mannheim
Wook-Shin Han, Postech
Thomas Heinis, Imperial College London
Arantza Illarramendi, U País Vasco
Alekh Jindal, MIT
Alfons Kemper, TU Munich
Martin Kersten, CWI
Arijit Khan, Nanyang Technological U
Georg Lausen, U Freiburg
Wang-Chien Lee, Penn State U
Wolfgang Lehner, TU Dresden
Chuan Lei, NEC Labs America
Hong Va Leong, Hong Kong Polytechnic U
Ulf Leser, HU Berlin
Guoliang Li, Tsinghua U
Eric Lo, Polytecnic U Hong Kong
Sofian Maabout, U Bordeaux
Patrick Marcel, U Tours
Sebastian Michel, TU Kaiserslautern
Eltabakh Mohammed, Worcester PI
Mohamed Mokbel, U Minnesota
Kjetil Nørvåg, Norwegian U Sc. & Techn.
Esther Pacitti, U Montpelier and INRIA
Marta Patino-Martinez, Politecnica Madrid
Torben Bach Pedersen, Aalborg U
Jian Pei, Simon Fraser U
Jean-Marc Petit, Université Lyon, CNRS
Peter Pietzuch, Imperial College London
Evaggelia Pitoura, U Ioannina
Pascal Poncelet, U Montpelier
Nicoleta Preda, U Versailles
Philippe Pucheral, U Versailles
Jorge-Arnulfo Quiane-Ruiz, Qatar CRI
Maya Ramanath, IIT Delhi
Rodolfo F. Resende, U Fed. Minas Gerais
Tore Risch, Uppsala U
Elke A. Rundensteiner, Worcester PI
Pierangela Samarati, Università Milano
Mohamed Sarwat, Arizona State U
Sebastian Schelter, Amazon
Marc H. Scholl, U Konstanz
Heiko Schuldt, U Basel
Timos Sellis, Swinburne U Techn.
Divesh Srivastava AT&T Labs-Research
Jianwen Su, UC Santa Barbara
Nan Tang, Qatar Comptg. Res. Inst.
Martin Theobald, U Antwerp
Farouk Toumani, U Clermont-Ferrand
Peter Triantafillou, U Glasgow
Vassilis Tsotras, UC Riverside
Yannis Velegrakis, U Trento
Daisy Wang, U Florida
Wei Wang, U New South Wales
Yinghui Wu, Washington State U
Xiaochun Yang, Northeastern U
Demetris Zeinalipour, U Cyprus
Xiaofang Zhou, The U Queensland
Esteban Zimanyi, U Libre Bruxelles

Industrial Program Committee

Nipun Agarwal, Oracle
Divy Agrawal, UC Santa Barbara
Stefan Dessloch, TU Kaiserslautern
Michael Gesmann, Software AG
Christoph Groeger, Bosch
Martin Grund, Amazon
Thomas Jörg, Google
Frank Leymann, U Stuttgart
Kostas Magoutis, U Ioannina
Albert Maier, IBM
Christian Mathis, SAP
Bernhard Mitschang, U Stuttgart
Mario Nascimento, U Alberta
Berthold Reinwald, IBM
Fabrizio Silvestri, Facebook
Eric Simon, SAP

Demonstration Program Committee

Ira Assent, Aarhus U
Carsten Binnig, Brown U
Angela Bonifati, Lille 1 Un
Isabel Cruz, U Illinois Chicago
Michael Gertz, Heidelberg U
Dirk Habich, TU Dresden
Sven Helmer, Free U Bolzano
Katja Hose, Aalborg U
Stefan Manegold, CWI Amsterdam
Anisoara Nica, SAP SE
Norman Paton, U Manchester
Heiko Schuldt, U Basel
Letizia Tanca, Politecnico Milano
Kian-Lee Tan, National U Singapore
Vassalos Vasilis, Athens U
Demetris Zeinalipour, MPI Informatics & U Cyprus

External Reviewers

Julien Aligon, Toulouse U
Mohand Boughanem, IRIT U Toulouse
Marc Bux, HU Berlin
Sean Chester, Norwegian U Sc. & Techn.
Dong-Wan Choi, Simon Fraser U
Lingyang Chu, Simon Fraser U
Mahfoud Djedaini, U Tours
Miguel Forte, Workable
Olga Gkountouna, IMIS/RC Athens
Aris Gloulalas-Divanis, IBM
Nicolas Labroche, U Tours
Mirjana Mazuran, Politecnico Milano
Xinghao Pan, UC Berkeley
Miguel Pardal, IST, U Lisbon
Emanuele Rabosio, Politecnico Milano
Patrick Schäfer, HU Berlin
George Stoilos, AUEB
Yu Yang, Simon Fraser U

Test-of-Time Award

In 2014, the Extended Database Technology conference (EDBT) began awarding the EDBT test-of-time (ToT) award, with the goal of recognising papers presented at EDBT Conferences that have had the most impact in terms of research, methodology, conceptual contribution, or transfer to practice.
This year, covering the conferences from 1996 to 2002, the award has been given to:

Mining Sequential Patterns:
Generalizations and Performance Improvements.
by Ramakrishnan Srikant, Rakesh Agrawal
published in the EDBT 1996 proceedings, 3–17.

This paper has made substantial contributions to data mining, and has had great influence on the work of others, as reflected by over 2900 citations on Google Scholar.
The paper formalizes a new variant of the problem of mining *sequential* patterns and develops and implements GSP, an algorithm to solve this problem. This paper extends the definition of sequence mining that was introduced by the same authors in a previous publication: Mining Sequential Patterns. ICDE 1995. The goal is to discover all sequential patterns with a user-specified minimum support from a database of sequences, where each sequence is a list of transactions ordered by transaction-time, and each transaction is a set of items. The proposed extensions are:

1. Time constraints: the authors generalised their previous definition of sequential patterns to admit max-gap and min-gap time constraints between adjacent elements of a sequential pattern.

2. Sliding windows: the authors relaxed the restriction that all the items in an element of a sequential pattern must come from the same transaction, and allowed a user-specified window-size within which the items can be present.

3. Taxonomies: the sequential patterns may include items across different levels of a taxonomy.

GSP guarantees that all rules that have a user-specified minimum support. It is shown to be much faster than the AprioriAll algorithm in the previous publication (on both synthetic and real data). GSP has been implemented as part of the Quest data mining prototype at IBM Research, and is incorporated in the IBM data mining product.
The EDBT 2017 Test of Time Award Committee consisted of Peter Triantafillou, Gustavo Alonso, Sihem Amer-Yahia, Ralf Hartmut Güting and Volker Markl.
The EDBT ToT award for 2017 will be presented during the EDBT/ICDT 2017 Joint Conference, March 21–24, in Venice, Italy (http://edbticdt2017.unive.it/).

Table of Contents

Research Papers

Poster Papers

Demonstrations

Tutorials

Industrial and Applications Papers

Parallel Array-Based Single- and Multi-Source Breadth First Searches on Large Dense Graphs

Moritz Kaufmann
Technical University of Munich
kaufmanm@in.tum.de

Manuel Then
Technical University of Munich
then@in.tum.de

Alfons Kemper
Technical University of Munich
kemper@in.tum.de

Thomas Neumann
Technical University of Munich
neumann@in.tum.de

ABSTRACT

One of the fundamental algorithms in analytical graph databases is breadth-first search (BFS). It is the basis of reachability queries, centrality computations, neighborhood enumeration, and many other commonly-used algorithms.

We take the idea of purely array-based BFSs introduced in the *sequential* multi-source MS-BFS algorithm and extend this approach to *multi-threaded single-* and *multi-source* BFSs. Replacing the typically used queues with fixed-sized arrays, we eliminate major points of contention which other BFS algorithms experience. To ensure equal work distribution between threads, we co-optimize work stealing parallelization with a novel vertex labeling. Our BFS algorithms have excellent scaling behavior and take advantage of multi-core NUMA architectures.

We evaluate our proposed algorithms using real-world and synthetic graphs with up to 68 billion edges. Our evaluation shows that the proposed multi-threaded single- and multi-source algorithms scale well and provide significantly better performance than other state-of-the-art BFS algorithms.

1. INTRODUCTION

Graphs are a natural abstraction for various common concepts like communication, interactions as well as friendships. Thus, graphs are a good way of representing social networks, web graphs, and communication networks. To extract structural information and business insights, a plethora of graph algorithms have been developed in multiple research communities.

At the core of many analytical graph algorithms are breadth first searches (BFSs). During a BFS, the vertices of a graph are traversed in order of their distance—measured in hops—from a source vertex. This traversal pattern can for example be used to do shortest path computations, pattern matchings, neighborhood enumerations, and centrality calculations. While all these algorithms are BFS-based, many different BFS variants have been published. Important aspects that differentiate BFS variants are their *degree of parallelism*, the *number of sources* they consider, and the *type of graph* they are suited for.

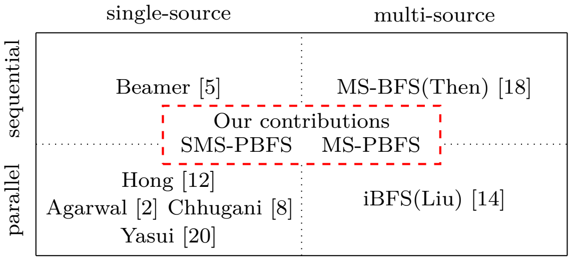


Figure 1: State-of-the-art single-server breadth-first-search publications

Possible degrees of parallelism include single-threaded and multi-threaded execution as well as distributed processing. Many emerging systems, e.g., Pregel [16], Spark [21], and GraphLab [15], focus heavily on distributed processing, but often neglect to optimize for the single-machine use case. However, especially for graph analytics, distributed processing is a very hard problem. The main reason for this is the high communication cost between compute nodes which is directly influenced by the inherent complexity of graph partitioning [4]. While there are cases in which distribution cannot be avoided, we argue that in graph analytics it is often done unnecessarily, leading to diminished performance. Actually, most—even large-scale—real-world graphs easily fit into the main memory of modern server-class machines [11]. Thus, we only consider the single-node scenario but differentiate between single and multi-threaded processing.

In Figure 1 we give an overview of the multi-threaded single-node state-of-the-art BFS algorithms.

The figure also includes the second important aspect of a BFS variant: its number of sources. Traditionally, the BFS problem is stated as traversing the graph from a single source vertex. While this single-source model can be applied to any BFS-based algorithm, it hampers inter-BFS optimizations. Specialized multi-source BFS algorithms like MS-BFS [18] and the GPU based iBFS [14] concurrently traverse the graph from multiple source vertices and try to share common work between the BFSs. This is, for example beneficial when the all pairs shortest path (APSP) problem needs to be solved as it is the case for the closeness centrality metric.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.02

For this computation, a full BFS is necessary from every vertex in the graph. Considering that small-world networks often consist of a single large connected component, a single-source BFS would visit every vertex in each traversal while a multi-source-optimized BFS batches visits where possible.

One central limitation of current multi-source BFS algorithms is their limited ability to analyze large graphs efficiently. The GPU-based iBFS is limited to the memory available on GPU cards which is over an order of magnitude less than what is available in modern servers. The CPU-based MS-BFS on the other hand is sequential; utilizing all cores would require an separate BFS instance for each CPU core. It can only speed up analysis when a huge number of sources is analyzed and it requires much more memory due to the separate BFS states.

In this paper we propose two breadth-first search algorithms that are optimized for modern massively-parallel multi-socket machines: SMS-PBFS and MS-PBFS. *SMS-PBFS* is a parallelized single-source BFS, while *MS-PBFS* is a parallelized multi-source BFS. Both algorithms are based on the approaches introduced by the sequential MS-BFS. By adding scalable parallelization we enable massive speedups especially when working with a limited number of sources. Our approach also significantly reduces memory requirements for parallelized multi-source BFSs.

Our evaluation using real-world graphs as well as artificial graphs, including the industry-standard benchmark Graph500 [1], shows that SMS-PBFS and MS-PBFS greatly outperform the existing state-of-the-art BFS algorithms. Because the overhead for parallelization is negligible, our parallelized algorithms can be efficiently used for sequential BFS traversals without modifications.

Specifically, the contributions of this paper are as follows:

- We present the *MS-PBFS* algorithm, a multi-core NUMA-aware multi-source BFS that ensures full machine utilization even for a limited number of sources. We also introduce *SMS-PBFS*, a multi-core NUMA-aware single-source BFS based on MS-PBFS that shows better performance than existing single-source BFS algorithms.
- We introduce a new vertex labeling scheme that is both cache-friendly as well as skew-avoiding.
- We propose a parallel low-overhead work stealing scheduling scheme that preserves NUMA locality in BFS workloads.

The latter two contributions can also boost the performance of existing BFS algorithms as well as other graph algorithms.

The paper is structured as follows. In Section 2 we describe the state-of-the-art BFS algorithms for the sequential and parallel single-source case as well as for the sequential multi-source case and summarize their limitations. Afterward, in Section 3 we present our novel algorithms MS-PBFS and SMS-PBFS. In Section 4 we describe our optimized scheduling algorithm, vertex labeling scheme, and memory-layout for modern NUMA architectures. Section 5 contains the evaluation of our algorithms. We give an overview over the related work in Section 6. Section 7 summarizes our findings.

2. BACKGROUND

In this section we describe the current state-of-the-art BFS algorithm variants. We focus on algorithms that operate on undirected, unweighted graphs. Such a graph is represented by a tuple $G = \{V, E\}$, where V is the set of vertices and $E = \{neighbors_v | v \in V\}$ where $neighbors_v$ is the set of neighbors of v. Additionally, we assume that the graphs of interest are *small-world networks* [3], i.e., that they are strongly connected and their number of neighbors per vertex follows a power law distribution. This is the case for most real-world graphs; examples include social networks, communication graphs and web graphs.

Given a graph G and a source vertex s, a BFS traverses the graph from s until all reachable vertices have been visited. During this process, vertices with a one-hop distance from s are visited first, then all vertices with distance two and so on. Each distance corresponds to one iteration. While executing an iteration the neighbors of vertices that were newly discovered in the previous iteration are checked to see if they have not yet been discovered. If so, they are marked as newly seen and enqueued for the next iteration. Consequently, the basic data structures during execution are a queue of vertices that were discovered in the previous iteration and must be processed in the current iteration, called *frontier*, a mapping *seen* that allows checking if a vertex has already been visited, and a queue *next* of vertices that were newly discovered in the current iteration. The latter queue is used as input for the next iteration. The number of BFS iterations corresponds to the maximum distance of any vertex from the source. It is bound by the diameter of the graph, i.e., the greatest shortest distance between any two vertices.

Our novel MS-PBFS and SMS-PBFS algorithms build on multiple existing techniques which we introduce in the following. We categorize the presented algorithms as either parallel or sequential, and as either single-source or multi-source as shown in Figure 1. To the best of our knowledge, each of the presented algorithms in this chapter is the current single-server state-of-the art in its category.

2.1 Sequential and Parallel Single-Source BFS

The fastest sequential single source BFS algorithm for dense graphs was presented by Beamer et al. [5]. It breaks up the algorithmic structure of the traditional BFS to especially reduce the amount of work required to analyze small-world networks. In such graphs most vertices are reached within few iterations [18]. This has the effect that at the end of this “hot phase” the frontier for the next iteration contains many more vertices than there are unseen vertices in the graph, as most were already discovered. At this point the classical *top-down* approach — trying to find unseen vertices by processing the *frontier* — becomes inefficient. Most vertices’ neighbors will already have been seen but would still need to be checked. The ratio of vertices discovered per traversed edge becomes very low. For these cases Beamer et al. propose to use a *bottom-up* approach and iterate over the vertices that were not yet seen in order to try to find an already seen vertex in their neighbor lists. Even though the result of the BFS is not changed, this approach significantly reduces the number of neighbors that have to be checked. This translates into better traversal performance, thus, this approach is often used in more specialized BFS algorithms [2, 18, 20].

Those algorithmic changes also have implications on the BFS data structures that can be used. Typically, the queues in a BFS are implemented using either a dense bitset or a sparse vector. The bottom-up phase, though, requires efficient lookups of vertices in the queue, thus, it can not be used efficiently with a sparse vector. The original authors solve this by converting the data structures from bitset to sparse vector when switching from top-down to bottom-up or vice versa.

For *parallelization*, this approach can be combined with existing work on scalable queues for BFSs [2, 12, 8, 5, 20] and on scalability on multi-socket NUMA architectures [19, 8] through static partitioning of vertices and data across NUMA nodes. Many of these techniques are combined in the BFS algorithm proposed by Yasui et al. [20, 19] which is the fastest multi-threaded single-source BFS.

2.2 Sequential Multi-Source BFS

The MS-BFS algorithm [18] is targeted towards multi-source traversal and further reduces the total number of neighbor lookups across all sources compared to Beamer et al. It is based on two important observations about BFS algorithms. Firstly, regardless the data structure used, for sufficiently large graphs it is expensive to check whether a vertex is already contained in *seen* as CPU cache hit rates decrease. For this very frequent operation even arrays with their containment check bound of $O(1)$ are bound by memory latency. This problem is further exacerbated on non-uniform memory access (NUMA) architectures that are common in modern server-class machines. Secondly, when multiple BFSs are run in the same connected component, every vertex of this component is visited *separately* in each BFS. This leads to redundant computations, because whenever two or more BFS traversals in the same component find a vertex v in the same distance d from their respective source vertices, the remainder of those BFS traversals from v will likely be very similar, i.e., visit most remaining vertices in the same distance.

MS-BFS alleviates some of these issues by optimizing for the case of executing multiple independent BFSs from different sources in the same graph. It uses three k-wide bitsets to encode the state of each vertex during k concurrent BFSs:

1. $seen[v]$, where the bit at position i indicates whether v was already seen during the BFS i,
2. $frontier[v]$, determining if v must be visited in the current iteration for the BFSs, and
3. $next[v]$, with each set bit marking that the vertex v needs to be visited in the following iteration for the respective BFS.

For example given $k = 4$ concurrent BFSs, the bitset $seen[v] = (1, 0, 0, 1)$ indicates that vertex v is already discovered in BFSs 0 and 3 but not in BFSs 1 and 2. Using this information, a BFS step to determine *seen* and *next* for all neighbors n of v can be executed using the bitwise operations *and* (&), *or* (|), and *negation* (∼):

```
for each $n \in neighbors[v]$
    $next[n] \leftarrow next[n] \mid (frontier[v] \;\&\; \sim seen[n])$
    $seen[n] \leftarrow seen[n] \mid frontier[v]$
```

Here, if n is not yet marked seen for a BFS and this BFS's respective bit is set in $frontier[v]$, then the vertex n is marked as seen and must be visited in the next iteration. The bitwise operations calculate *seen* and *next* for k BFSs at the same time and can be computed efficiently by leveraging the wide registers of modern CPUs. A full MS-BFS iteration consists of executing these operations for all vertices v in the graph. Note that all k BFSs are run concurrently on a single CPU core with their traversals implicitly merged whenever possible.

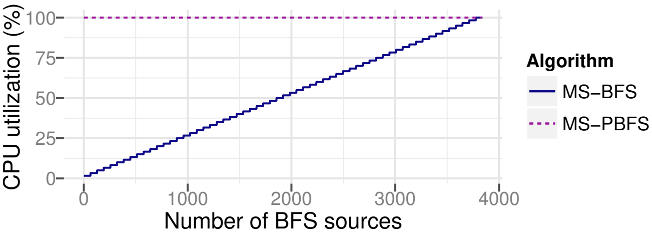


Figure 2: CPU utilization of MS-BFS and MS-PBFS as the number of sources increases.

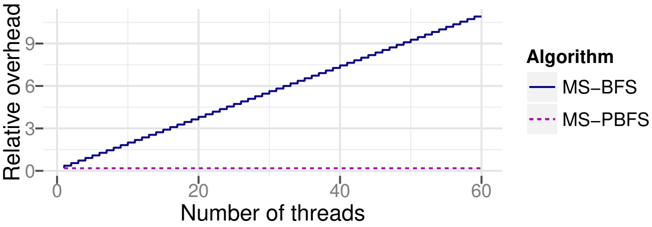


Figure 3: Relative memory overhead compared to graph size as number of threads increases.

MS-BFS works for any number of concurrent BFSs using bitset sizes chosen accordingly. However, it is especially efficient when the vertex bitsets have a width for which the target machine natively supports bit operations. Modern 64-bit x86 CPUs do not only have registers and instructions that support 64 bit wide values, but also ones for 128 bit and 256 bit using the SSE and AVX-2 extensions, respectively. The original publication elaborates on the trade-offs of various bitset widths and how they influence the algorithm's performance.

In Section 3 we show how MS-BFS can be efficiently parallelized and present an optimized variant that is highly efficient for single source traversals.

2.3 Limitations of Existing Algorithms

The MS-BFS algorithm is limited to sequential execution. The only way to saturate a multi-core system is to run a separate MS-BFS instance on each core. However, if the number of BFS sources is limited, e.g., to only 64 as in the Graph500 benchmark, MS-BFS can only run single-threaded or, at best, on few cores. In such cases, the capabilities of a multi-core system cannot be fully utilized. Figure 2 analyzes this problem using a 60-core machine and 64 concurrent BFSs per MS-BFS. Every 64 sources one more thread can be used. Hence, only with 3840 or more sources all cores are utilized. Also, by running multiple sequential instances simultaneously, the memory requirements rise drastically to the point that the dynamic state of the BFSs require much more memory than the graph itself. This is demonstrated in Figure 3. It compares the memory required for the MS-BFS and our proposed MS-PBFS data structures to the size of the analyzed graph. We calculated the memory requirement based on 16 edges per vertex like the Kronecker graphs in the Graph500 benchmark. While traditional BFSs only require a fraction of the graph memory for their working set, MS-BFS

Listing 1: Top-down MS-BFS algorithm from [18].

```
1  for each $v \in V$
2      if $frontier[v] = \varnothing$, skip
3      for each $n \in neighbors_v$
4          $next[n] \leftarrow next[n] \mid frontier[v]$
5
6  for each $v \in V$
7      if $next[v] = \varnothing$, skip
8      $next[v] \leftarrow next[v] \;\&\; \sim seen[v]$
9      $seen[v] \leftarrow seen[v] \mid next[v]$
10     if $next[v] \neq \mathbb{B}_\varnothing$
11         $v$ is found by BFSs in $next[v]$
```

already requires more memory than the graph using only 6 threads. With 60 threads it requires over 10 times more memory! Hence, more than one terabyte of main memory would be needed to analyze a 100GB graph using all cores. An alternative could be to use smaller batch sizes, thus, requiring fewer sources and memory to take advantage of all cores. However, that would decrease the traversal performance as less work can be shared between the BFSs. In contrast, the parallel multi-source algorithm MS-PBFS proposed in this paper can use all cores at 64 BFSs and only consumes as much memory as a single MS-BFS.

State-of-the-art parallel single-source algorithms are limited by either locality and scalability issues associated with the sparse queues. Even if partitioned at NUMA socket granularity there can be a lot of contention and the trend of having more cores per CPU socket does not work in such approaches favor.

3. PARALLEL SINGLE- AND MULTI-SOURCE BFS

In this section we present our *parallelized* multi-source BFS algorithm as well as a single-source BFS variant, both designed to avoid those problems.

3.1 MS-PBFS

In the following we introduce MS-PBFS, a parallel multi-source BFS algorithm that ensures full machine utilization even for a single multi-source BFS.

MS-PBFS is based on MS-BFS and parallelizes both its top-down (Section 3.1.1) and its bottom-up (Section 3.1.2) variant. Our basic strategy is to parallelize all loops over the vertices by partitioning them into disjunct subsets and processing those in parallel. State that is modified and accessed in parallel then has to be synchronized to ensure correctness.

3.1.1 Top-down MS-PBFS

MS-BFS uses a two-phase top-down variant, shown in Listing 1. As described in the Section 2, each value in *seen*, *frontier* and *next* is not a single boolean value but a bitset. The first phase, lines 1 through 4, aggregates information about which vertices are reachable in the current iteration. After it finishes, the second phase, the loop in lines 6 through 11, identifies which of the reachable vertices are newly discovered and processes them.

Our strategy is to parallelize both of these loops and separate them using a barrier. During this parallel processing, the first loop accesses the fixed-size *frontier*, *neighbors* and *next* data structures. As the former two are constant during the loop accesses, they do not require any synchronization. The *next* data structure on the other hand is updated for each neighbor n by combining n's *next* bitset with the currently visited v's *frontier*. As vertices in general can be reached via multiple edges from different vertices, different threads might update *next* simultaneously for a vertex. To avoid losing information in this situation, we use an atomic compare and swap (CAS) instruction, replacing line 4 with the following:

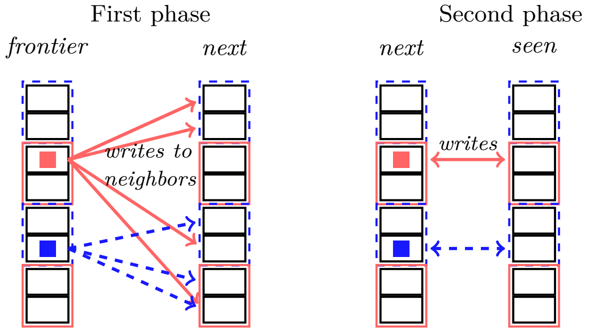


Figure 4: Concurrent top-down memory accesses

```
do
    $oldNext \leftarrow next[n]$
    $newNext \leftarrow oldNext \mid frontier[v]$
while $atomic\_cas(next[n], oldNext, newNext)$
```

Bitsets wider than the CPU's largest atomic operation value type can be supported by implementing the update operation as a series of independent atomic CAS updates of each sub-part of the bitset. For example a 512-bit bitset could be updated using eight 64-bit CAS as described above. This retains the desired semantics as the operation can only add bits but never unset them. It is also not required to track which thread first added which bit as the updates of the newly discovered vertices is only done in the second phase.

The second phase iterates over all vertices in the graph and updates *next* and *seen*. In contrast to the first phase, no two worker threads can access the same entry in the data structures. Regardless of how the vertex ranges are partitioned, there is a bijective mapping between a vertex, the accessed data entries, and the worker that processes it. Consequently, there cannot be any conflicts, thus, no synchronization is necessary.

Figure 4 visualizes the memory access patterns for all writes in the first and second phase of the top-down MS-PBFS algorithm. The example shows a configuration with two parallel workers and a task size of two. Squares show the currently active vertices and arrows point to entries that are modified. The linestyle of the square shows the association to the different workers. We come back to this figure in Section 4.4 to further explain the linestyles.

To reduce the time spent between iterations, we directly clear each *frontier* entry inside the second parallelized loop. This allows MS-PBFS to re-use the memory of the current *frontier* for *next* in the subsequent iteration without having to clear the memory separately. Thus, we reduce the algorithm's memory bandwidth consumption.

Furthermore, we only update *next* entries if the computation results in changes to the bitset. This avoids unnecessary writes and cache line invalidations on other CPUs [2].

Listing 2: Bottom-up MS-BFS traversal from [18].

```
for each $u \in V$
  if $|seen[u]| = |S|$, skip
  for each $v \in neighbors_u$
      $next[u] \leftarrow next[u]$ | $frontier[v]$
  $next[u] \leftarrow next[u]$ & $\sim seen[u]$
  $seen[u] \leftarrow seen[u]$ | $next[u]$
  if $|next[u]| \neq 0$
      $u$ is found by BFSs in $next[u]$
```

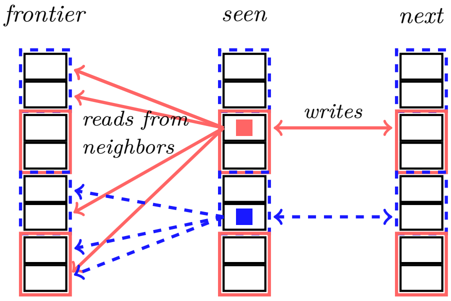


Figure 5: Concurrent bottom-up memory accesses

3.1.2 Bottom-up MS-PBFS

As explained in Section 2.1, a BFS's bottom-up variant linearly traverses the *seen* data structure to find vertices that have not been marked yet. For every vertex v that is not yet seen in all concurrent BFSs, MS-BFS's bottom-up variant checks whether any of its neighbors was already seen in the respective BFS. If so, v is marked as seen and is visited in the next iteration. We show the full bottom-up loop in Listing 2.

MS-PBFS parallelizes this loop by splitting the graph into distinct vertex ranges which are then processed by worker threads. Inside the loop, the current iteration's *frontier* is only read. Both *seen* and *next* are read as well as updated. Similar to the second phase described in the previous section, there is a bijective mapping between each updated entry and the worker that processes the respective vertex. Consequently, there cannot be any read-write or write-write conflicts and, thus, no synchronization is required within the ranges. Figure 5 depicts the bottom-up variant's memory access pattern, again for two parallel workers.

Once all active BFSs bits are set in *next* we stop checking further neighbors to avoid unnecessary read operations. This check is also used in the original bottom-up algorithm by Beamer et al.

3.2 Parallel Single-Source: SMS-PBFS

In order to also apply our novel algorithm to BFS that traverse the graph from only a single source, we derive a single-source variant: SMS-PBFS. SMS-PBFS contains two main changes: the values in each array are represented by boolean values instead of bitsets, and checks that are only required when multiple BFS are bundled can be replaced by constants. This allows us to simplify the atomic update in the top-down algorithm, as a single atomic write is sufficient, instead of a compare and swap loop. The SMS-PBFS top-down and bottom-up algorithms are shown in Listing 3 and 4, respectively. Parallel coordination is only required when scheduling the vertex-parallel loops and during the single atomic update in the first top-down loop.

Listing 3: Single-source parallel top-down algorithm

```
parallel for each $v \in V$
  if not($frontier[v]$), skip
  for each $n \in neighbors_v$
    if not($next[n]$), atomic($next[n] \leftarrow true$)
  $frontier[v] \leftarrow false$

parallel for each $v \in V$
  if not($next[v]$),  skip
  $next[v] \leftarrow$ not($seen[v]$)
  if not($seen[v]$)
    $seen[v] \leftarrow true$
    $v$ is found
```

Listing 4: Single-source parallel bottom-up algorithm

```
parallel for each $u \in V$
  if $seen[u]$
    $next[u] \leftarrow false$
  else
    for each $v \in neighbors_u$
      if $frontier[v]$
        $next[u] \leftarrow true$
        break
    if $next[v]$
        $seen[u] \leftarrow true$
        $u$ is found
```

While MS-PBFS always has to use an array of bitsets to implement *next*, *frontier* and *seen*, there is more freedom when implementing SMS-PBFS. It is still restricted to using dense arrays, but each entry can either be a bit, a byte or a wider data type. In the parallel case, where the state of 512 vertices fits into one 64-byte CPU cache line using bit representation, the chance of concurrent modification is very high. Choosing a larger data type allows to balance cache efficiency and reduced contention between workers. We demonstrate these effects in our evaluation. To reduce the number of branches, we try to detect when a consecutive range of vertices is not active in the current iteration and skip it. Instead of checking each vertex individually we check ranges of size 8 bytes, which can efficiently be implemented on 64-bit CPUs. Using a bit representation, each such range contains the status of 64 vertices. If no bit is set, we directly jump to the next chunk and save a large number of individual bit checks. Otherwise, each vertex is processed individually. This is similar to the *Bitsets-and-summary* optimization [19] but does not require an explicit summary bit.

4. SCHEDULING AND PARALLEL GRAPH DATA STRUCTURES

The parallelized algorithms' descriptions in Section 3 focus on how to provide semantical correctness. It leaves out the implementation details of how to actually partition the work to workers and how to store the BFS data structures as well as the graph. As shown by existing work on parallel single-source BFSs, these implementation choices can have a huge influence on the performance of algorithms that are intended to run on multi-socket machines with a large number of cores. In this section we describe the data structures and memory

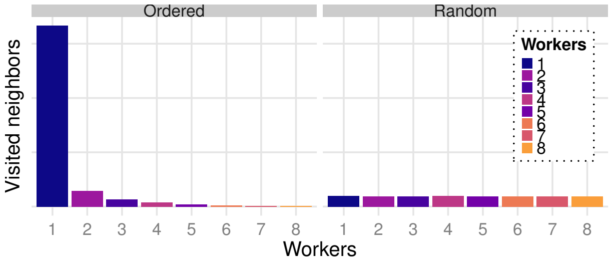


Figure 6: Visited neighbors per worker during a BFS using static partitioning on a social network graph with different vertex labelings

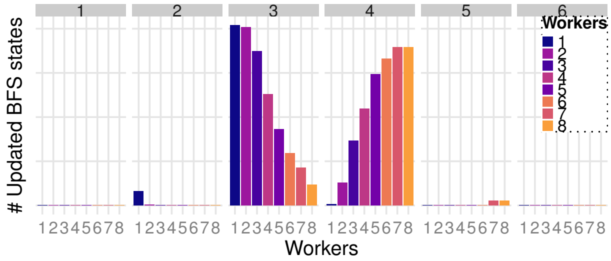


Figure 7: Updated BFS vertex states per worker per iteration during a BFS using static partitioning on a social network graph with ordered vertex labeling

organization to efficiently scale MS-PBFS and SMS-PBFS (together abbreviated as (S)MS-BFS) on such machines.

4.1 Parallelization Strategies

The initial version of our parallelized algorithms used popular techniques from state-of-the-art parallel single-source implementations. Specifically, it used static partitioning of vertices to workers, and degree-ordered vertex labeling[19]. With this labeling scheme, we re-labeled the graph's vertices and assigned dense ids in the order of the vertices' degrees, with the highest-degree vertex getting the smallest id. That way, the states of high degree vertices are located close together which improves cache hit rates. This first implementation showed very good performance for a small number of workers. However, at higher thread counts, the overall scalability was severely limited.

Together with static partitioning, degree based labeling has the effect that due to the power-law distribution of vertex degrees in many real world graphs, the vertices in the first partitions have orders of magnitude more neighbors than those in later partitions. We visualize this effect in Figure 6. In that experiment, the first worker processes the first $\frac{1}{8}$th of the vertices in the graph, the second worker the second $\frac{1}{8}$th, and so on. As the amount of work per partition increases with the number of neighbors that need to be visited, the described skew directly affects the workers' runtime, as it is one of the most costly operations besides updating the BFS vertex state. While it may be possible to create balanced static partitions such that each worker has to do the same amount of work across all BFS iterations, it is not enough to significantly increase utilization. The problem would then be that in different iterations different parts of the graph are active, thus, there would still be a large runtime skew in each iteration. The different workload for workers across iterations is shown in Figure 7 using the number of updated BFS vertex states as an indicator for the actual amount of work.

Additionally, this figure gives an indication why dynamic work assignment does not ensure full utilization on its own. Intuitively, in a small-world network an average BFS traverses the graph starting from the source vertex to the vertices with the highest degrees, because these are well-connected, and from there to the remainder of the graph. For such a BFS, the high-degree vertices are typically discovered after two to three iterations as shown in Figure 7. There in iteration two only a tiny fraction of vertices is updated. On the other hand as these are the high-degree vertices a lot of undiscovered vertices are reachable from them, resulting in a huge number of updates in iteration three. The updates themselves, which are processed in the second phase of the top-down algorithm could be well distributed across workers. Identifying the newly reachable vertices, which is done in the first phase, by searching the neighbors of the high-degree vertices is more challenging to schedule because there are only few and due to the labeling they are all clustered together. In combination, this iteration is very expensive but a large part of the work is spent when processing very few high-degree vertices. To achieve even utilization, tiny task sizes would be required. Such tiny tasks mean, however, that the scheduling overhead would become so significant that the overall performance would not improve.

Instead, our design relies on two strategies. We use fine-granular tasks together with work-stealing to significantly reduce the number of vertices that are assigned at once and enable load-balancing between the cores. We also use a novel vertex labeling scheme that is scheduling-aware and distributes high-degree vertices in such a way that they are both clustered but also spread across multiple tasks. This allows us to avoid the use of tiny task ranges.

4.2 Parallel Vertex Processing

In this section we focus on providing a parallelization scheme that minimizes synchronization between threads and balances work between nodes to achieve full utilization of all cores during the whole algorithm.

Our concept allows load balancing through work stealing with negligible synchronization overhead.

4.2.1 Task creation and assignment

Efficient load balancing requires tasks to have two related properties: there need to be many tasks and their runtime needs to be relatively short compared to the overall runtime. If there were only two tasks on average per thread, a scenario is very probable where a slow thread only starts its last task when all other threads are already close to being finished with their work. This creates potential to have all other threads idling until this last thread is finished. Given a fixed overall runtime, the shorter the runtime of each work unit and the more tasks are available, the easier it becomes to get all threads to finish at approximately the same time. On the other hand, if the ranges are very small, the threads have to request tasks more often from the task pool. This can lead to contention and, thus, decrease efficiency because more processing time is spent in scheduling instead of doing actual work.

Due to the fixed-size *frontier* and *next* arrays which span

Listing 5: Task creation algorithm: create_tasks

```
1  Input: $queueSize, splitSize, numThreads$
2  $workerTasks \leftarrow \varnothing$
3  $curWorker \leftarrow 0$
4  for($offset = 0; offset < queueSize; offset+ = splitSize$)
5      $wId \leftarrow curWorker$ mod $numThreads$
6      $range \leftarrow \{offset, \texttt{min}(offset + splitSize, queueSize)\}$
7      $workerTasks[wId] \leftarrow workerTasks[wId] \cup range$
8      $curWorker \leftarrow curWorker + 1$
9  $taskQueues \leftarrow \varnothing$
10 for $i = 1, \ldots, num\_threads$
11     $taskQueues[i] \leftarrow \{|workerTasks[i]|, workerTasks[i]\}$
12 return $taskQueues$
```

all the vertices no matter how many of them are actually enqueued, all parallel loops of (S)MS-PBFS can follow the same pattern: a given operation has to be executed for all vertices in the graph. To create the tasks we divide the list of vertices into small ranges. In our experiments we found that task range sizes of 256 or more vertices do not have any significant scheduling overhead (below 1% of total runtime) for a graph with more than one million vertices. With about 3900 tasks in such a graph there are enough tasks to load balance even machines with hundreds of cores.

For work assignment we do not use one central task queue, but similar to static partitioning give each worker its own queue. When a parallelized loop over the vertices of the graph is executed, the queues are initialized using the *create_tasks* function shown in Listing 5. Each task queue $taskQueues[i] = \{curTaskIx, queuedTasks\}$ belonging to worker i consists of an index *curTaskIx* pointing to the next task and a list of tasks *queuedTasks*. The number of vertices per task is controlled by the parameter *splitSize*. We use a round-robin distribution scheme, so the difference in queue sizes can be at most one task.

4.2.2 Work stealing scheduling

The coordination of workers during task execution is handled by the lock-free function *fetch_task*, which is shown in Listing 6. The function can be kept simple due to the fact that during a phase of parallel processing no new tasks need to be added. Only after all tasks have been completed the next round of tasks is processed—e.g., a new iteration or the second phase of the top-down algorithm.

Initially, each worker fetches tasks from its own, local queue which is identified using the *workerId* parameter. It atomically fetches and increments the current value of *curTaskIx* as shown in line 5. Using modern CPU instructions, this can be done without explicit locking. If the task id is within the bounds of the current task queue (line 7), the corresponding task range is processed by the worker. Otherwise, it switches to the next worker's task queue by incrementing the task queue offset, and tries again to fetch a task. This is repeated until either a task is found or, alternatively, after all queues have been checked, an empty task range is returned to the worker to signal that no task is available anymore. Further optimizations like remembering the task queue index where the current task was found and resuming from that offset when the next task is fetched, guarantee that every worker skips each queue exactly once. Incrementing the *curTaskIx* only if the queue is not empty avoids atomic writes which could lead to cache misses when other workers are visiting that queue.

Listing 6: Task retrieval algorithm: fetch_task

```
1  Input: $taskQueues, workerId$
2  $offset \leftarrow 0$
3  do
4      $i \leftarrow (threadId + offset\ )$mod $|taskQueues|$
5      $taskId \leftarrow fetch\_add\_task\_ix(taskQueues[i], 1)$
6      if $taskId < num\_tasks(taskQueues[i])$
7          return $get\_task(taskQueues)[i]$
8      else
9          $offset \leftarrow offset + 1$
10 while $offset < |taskQueues|$
11 return $empty\_range()$
```

Listing 7: Parallelized *for* loop

```
1  $tasks \leftarrow create\_tasks(|V|, splitSize, |workers|)$
2  run on each $w \in workers$
3      $workerId \leftarrow getWorkerId()$
4      while(($range \leftarrow fetch\_task(tasks, workerId)) \neq \varnothing$)
5          for each $v \in range$
6              $\{Loop\ \ body\}$
7  $wait\_until\_all\_finished(workers)$
```

Listing 7 shows how the task creation and fetching algorithms can be combined to implement the parallel *for* loop which is used to replace the original sequential loops in the top-down and bottom-up traversals. Here, *workers* is the set of parallel processors. In line 2 all workers are notified that new work is available and given the current *task* queues. Each worker fetches tasks and loops over the contained vertices until all tasks are finished. Once a worker can not retrieve further tasks it signals the main thread, which waits until all workers are finished.

As long as a worker only fetches from its own queue, the task retrieval cost is minimal—mostly only an atomic increment which is barely more expensive than a non-atomic increment on the x86 architecture[17]. Even when reading from remote queues, our scheduling has only minimal cost that mostly results from reading one value if the respective queue is already finished, and one write when fetching a task. This is negligible compared to the normal BFS workload of at least one atomic write per vertex in the graph. The construction cost of the initial queues in the *create_tasks* function is also barely measurable and could be easily parallelized if required.

4.3 Striped vertex order

In the introduction of Section 4.1, we discussed that the combination of multi-threading, degree ordered labeling and array-based BFS processing leads to large skew between worker runtimes. As (S)MS-PBFS's top-down algorithms is designed for multi-threading and requires efficient random-access to the frontier, neither the threading model, nor the backing data structure must be changed. Thus, though our single-threaded benchmarks confirmed that the increased cache locality achieved through degree-ordered labeling leads to significantly shorter runtimes compared to random vertex labeling, we cannot use degree-ordered labeling.

Instead, we propose a cache-friendly semi-ordered approach: We distribute degree-ordered vertices in a round robin fashion across the workers' task ranges. The highest-degree vertex is labeled such that it comes at the start of the first task of worker one. The second-highest degree vertex is labeled such that it comes at the start of the first task of worker two, etc. This round robin distribution is continued until all the first tasks for the workers are filled. Then, all the second tasks, and so on, until all vertices are assigned to the workers. Using this approach we still cannot guarantee that all task ranges have the same cost, but we can control that the cost of the ranges in each task queue are approximately the same per worker. Also, because the highest degree vertices are assigned first, the most expensive tasks will be executed first. Having small task sizes at the end has the advantage of reducing wait times towards the end of processing when no tasks are available for stealing anymore. The pre-computation cost of this *striped vertex labeling* is similar to that of degree-ordered labeling.

4.4 NUMA Optimizations

Our (S)MS-PBFS algorithms, as described above, scale well on single-socket machines. When running on multi-socket machines, however, the performance does only improve insignificantly, even though the additional CPUs' processing power is used. The main problems causing this are twofold. First, if all data is located in a single socket's local memory, i.e., in a single NUMA region, reading it from other sockets can expose memory bandwidth limitations. Second, writing data in a remote NUMA region can be very expensive [8]. This leads to a situation where the scalability of seemingly perfectly parallelizable algorithms is limited to a single socket. In the following, we describe (S)MS-PBFS optimizations that substantially improve the algorithms' scaling on NUMA architectures.

In our (S)MS-PBFS algorithms it is very predictable which data is read, particularly within a task range. Consider a bottom-up iteration as described in Section 3.1.2. When processing a task, it updates only the information of vertices inside that task. We designed our algorithm with the goals of not only providing NUMA scalability but *also* of avoiding any overhead from providing this scalability. We deterministically place memory pages for all BFS data structures, e.g., for *seen*, in the NUMA region of the worker that is assigned to vertices contained in the memory page. Further, we pin each worker thread to a specific CPU core so that it is not migrated during traversals. The desired result of this pinning is visualized in Figures 4 and 5. In addition to the figures' already discussed elements, we use the linestyle to encode the NUMA region of both the data and the workers. The memory pages backing the arrays are interleaved across the NUMA nodes at exactly the task range borders—for the example shown in the figures there are two vertices per task—and the workers only process vertices with data on the same NUMA node except for stolen tasks.

We calculate the mapping of vertices to memory pages and the size of task ranges as follows. Consider that a 64 bit wide bitset is used per *seen* entry, and memory pages have a common size of 4096 bytes. In this example, the task range size should be a multiple of $\frac{pageSize}{bitsetSize/8} = 512$ vertices. Given a task range, it is straightforward to calculate the memory range of the data belonging to the associated vertices.

Because we initialize large data structures like *seen*, *frontier*, and *next* only once at the beginning of the BFS and use them across iterations, we need to make sure that the data is placed deterministically, and that tasks accessing the same vertices are scheduled accordingly in all iterations. Thus, work stealing must not occur during the parallel initialization of the data structures to ensure proper initial NUMA region assignments of the memory pages.

When the BFS tasks only update memory regions that were initialized by themselves, we achieve NUMA locality. Note that even though our work stealing scheduling approach results in additional flexibility regarding task assignment, most tasks are still executed by their originally assigned workers when the total runtime for the tasks in each queue is balanced.

While this goal is perfectly attainable for the bottom-up variant and the second loop of top-down processing, we cannot efficiently predict which vertex information is updated in the first top-down loop. Besides processing stolen tasks, this is the *only* part of our algorithm in which non-local writes can happen.

In summary, given that each worker thread initializes the memory pages that correspond to the vertex ranges it is assigned to, nearly all write accesses, except for the first top-down loop and the work stolen from other threads, are NUMA-local. (S)MS-PBFS further guarantee that the share of memory located in each NUMA region is proportional to the share of threads that belong to that NUMA region. If, for example, 8 threads are located in NUMA region 0 and 2 threads are located in NUMA region 1, 80% of the memory required for the BFS data structures are located in region 0 and 20% will be in region 1.

Similar to the NUMA optimizations of the BFS data structures, also the graph storage can be optimized. We minimize cross-NUMA accesses by allocating the neighbor lists of the vertices processed in each task range on the same NUMA node as the worker which the task is assigned to. By using the same vertex range assignment while loading the neighbor lists and during BFS traversal, we ensure that all the data entries for each vertex are co-located. This principle is similar to the G^B partitioning described by Yasui et al. [19], which, however, uses static partitioning with one partition per NUMA node.

5. EVALUATION

In our evaluation we analyze four key aspects:

- What influence do the different labeling schemes have?
- How does the sequential performance of SMS-PBFS's algorithmic approach compare to Beamer's direction-optimizing BFS?
- How effectively does it scale both in terms of number of cores and dataset size?
- How does it compare to MS-BFS and the parallel single-source BFS by Yasui et al.?

In addition to the MS-PBFS and SMS-PBFS algorithms described before, we test two more variants. *MS-PBFS (sequential)* is our novel MS-PBFS algorithm run the same way as MS-BFS: a single instance per core requiring multiple batches to be evaluated in parallel. This tests the impact of the early exit optimization in the bottom-up phase, as well as our optimized data-structures. Another variant, *MS-PBFS (one per socket)* runs one parallel multi-source BFS per CPU socket using MS-PBFS. We use the performance of this variant to determine the cost of parallelization across all NUMA nodes when using MS-PBFS. Furthermore, SMS-PBFS is run in two variants: *SMS-PBFS (byte)* uses an array of bytes for *seen*, *frontier*, and *next*, and *SMS-PBFS (bit)* uses an array of bits.

Our test machine is a 4-socket NUMA system with 4x Intel Xeon E7-4870 v2 CPUs @ 2.3 GHz with one terabyte of main memory. Across all four CPUs, the system has 60 hardware threads. In the experiments we also used all Hyper-Threads.

We used both synthetic as well as real-world graphs. The synthetic graphs are Kronecker graphs [13] with the same parameters that are used in the Graph500 benchmark. They exhibit a structure similar to many large social networks. Additionally, for validation we also use artificial graphs generated by the LDBC data generator [9]. The generated LDBC graphs are designed to match the characteristics of real social networks very closely. Our used real-world graphs are chosen to cover different domains with various characteristic: the twitter follower graph, the uk-2005 web crawl graph and the hollywood-2011 graph describing who acted together in movies. The uk-2005 and hollywood-2011 graph were provided by the WebGraph project [7]. Table 1 lists the properties of all graphs used in our experiments. For the Kronecker graph we omit some of the in-between sizes as they always grow by a factor of 2. The vertex counts only consider vertices that have at least one neighbor. KG0 is a special variation of the Kronecker graph that was used in the evaluation of [14]; it was generated using an average out-degree of 1024.

The listed memory size is based on using 32-bit vertex identifiers and requiring $2 * vertex\_size = 8$ bytes per edge. To measure the performance of MS-BFS we use the source code published on github[1]. For comparison with Beamer, we used their implementation provided as part of the GAP Benchmark Suite (GAPBS) [6]. We did not have access to an implementation of Yasui et al. or iBFS; instead, we compare to published numbers on a similar machine using the same graph.

Our basic metric for comparison is the edge traversal rate (GTEPS). The Graph500 specification defines the number of traversed edges per source as number of input edges contained in the connected component which the source belongs to. Compared to the runtime which increases linearly with the graph size, this metric it is more suitable to compare performance across different graphs. In the MS-BFS paper, the number of edges was calculated by summing up the degrees of all vertices in the connected component. In the official benchmark, however, each undirected edge is only counted once. We use this method in our new measurements. To compare the numbers in this paper with the number of the MS-BFS paper, the other numbers have to be divided by two. In order to give an intuition about the effort required to analyze a specific graph we also show the time MS-PBFS requires for processing 64 sources in Table 1.

5.1 Labeling Comparison

To evaluate the different labeling approaches, we ran both MS-PBFS and SMS-PBFS using 120 threads on a scale 27 Kronecker graph with work stealing scheduling. The three tested variants are random vertex labeling (random), degree-ordered labeling (ordered), and our proposed striped vertex labeling (striped). The average runtime per BFS iteration for each scheme and algorithm is shown in Figure 8. Our results show that degree-ordered labeling exhibits significantly better runtimes that random labeling for the MS-PBFS algorithm. Especially in the most expensive third iteration, the difference between the approaches is close to a factor of two. This supports the results of existing work about graph re-labeling[19].

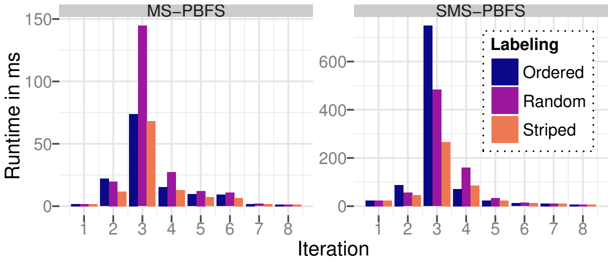


Figure 8: Runtime of BFS iterations under different vertex labeling strategies using SMS-PBFS and MS-PBFS.

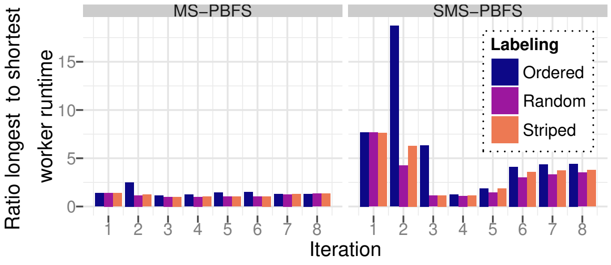


Figure 9: Skew in worker runtimes per iteration when running MS-PBFS and SMS-PBFS with different vertex labelings.

In contrast, for our array-based parallel single-source SMS-PBFS, random ordering exhibits better runtimes. Here, the skew, described in Section 4, and its related problems show their full impact. We evaluated this further in Figure 9 which shows the runtime difference between the longest to the shortest worker per iteration for all three labeling approaches. We see that skew is a much larger problem for SMS-PBFS than for MS-PBFS. Especially in the costly third iteration, there is a significant difference—more than factor 15 for degree-ordered—between *worker* runtimes per iteration. In MS-PBFS skew is a smaller problem as a much larger number of vertices is active in each iteration as there are so many BFSs active at once. Our novel striped vertex ordering shows the best overall runtimes and also balances the workload well. It combines the benefits of degree-based and random ordering in SMS-PBFS and MS-PBFS, while avoiding the other labelings' disadvantages. Similar to random labeling, striped vertex ordering provides good skew resilience, and like degree-ordering, it achieves very good cache locality. Using SMS-BFS the overall runtimes per BFS were: 42ms (striped), 86ms (ordered), 68ms (random).

5.2 Sequential Comparison

In this section, we analyze SMS-PBFS in a sequential setting and compare it against Beamer et al.'s state-of-the-art in sequential single-source BFSs. In addition to Beamer's GAPBS implementation, we also implemented two variants of their BFS that use the same graph, data structure and

[1] https://github.com/mtodat/ms-bfs

Name	Nodes (x10^6)	Edges (x10^6)	Memory size (GB)	MS-PBFS (runtime per 64)	MS-PBFS (GTEPS)	MS-BFS (GTEPS)	MS-BFS 64 (GTEPS)	SMS-PBFS (GTEPS)
Kronecker 20	2^{20}	15.7	0.119	3.27 ms	**307**	160	4.44	56.2 (bit)
Kronecker 26	2^{26}	1,050	7.96	246 ms	**274**	65.8	2.23	58.9 (bit)
Kronecker 32	2^{32}	68,300	5q5	39,700 ms	**110**	*failed (OOM)*	0.845	76.7 (bit)
KG0	0.982	364	2.72	12.5ms	**1860**	241	11.2	110 (bit)
LDBC 100	1.61	102	0.764	24.4 ms	**267**	76.6	3.01	39.2 (byte)
LDBC 1000	12.4	1010	7.61	551 ms	**118**	45.5	1.30	83.2 (byte)
Hollywood-2011	1.99	114	0.860	49.8 ms	**147**	59.6	2.19	26.5 (byte)
UK-2005	39.5	783	5.98	2220 ms	**22.6**	13.2	0.773	4.96 (bit)
Twitter	41.7	1,200	9.11	934 ms	**82.4**	32.5	1.13	21.0 (bit)

Table 1: Graphs description and algorithm performance in GTEPS using 60 threads.

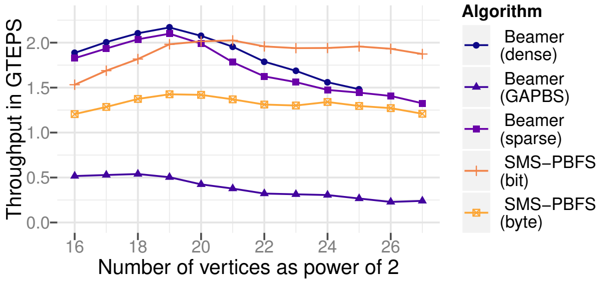


Figure 10: Performance of single-threaded BFS runs over varying graph sizes

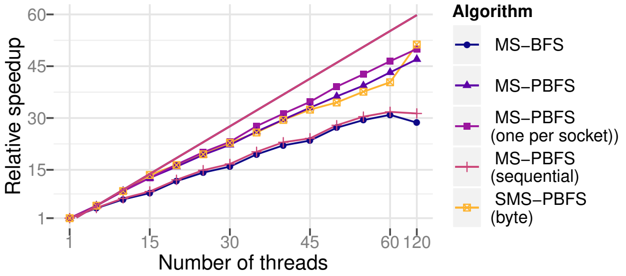


Figure 11: Relative speedup as number of threads increase in a 2^{26} vertices Kronecker graph

chunk skipping optimizations which we use for SMS-PBFS (bit). In the first variant, the queues in the top-down phase are backed by a sparse vector, and in the second variant we used a dense bit array. Both variants use the same bottom-up implementation.

Figure 10 shows the single-source BFSs throughput on a range of Kronecker graphs. The measurements show that for graphs with as few as 2^{20} vertices, our SMS-PBFS is already faster than Beamer et al.'s BFS. As the graph size increases, the probability that the data associated with a vertex is in the CPU cache decreases. There, our top-down approach benefits from having fewer non-sequential data accesses. On the other hand, our BFS has to iterate over all vertices twice. At small graph sizes this overhead can not be recouped as the reduction of random writes does not pay off when the write locations are in the CPU cache. For larger graph sizes, the improvement of SMS-PBFS over our Beamer implementation is limited, as the algorithms only differ in the top-down algorithms but a majority of the runtime in each BFS is spent in the bottom-up phase.

5.3 Parallel Comparison

In this section we compare our (S)MS-PBFS algorithms against the MS-BFS algorithm in a multi-threaded scenario. Inspired by the Graph500 benchmark, we fix the size of a batch for all algorithms to at most 64 sources. The MS-BFS algorithm is sequential and can only utilize all cores by running one algorithm instance per core. Thus, it requires at least $batch\_size * num\_threads = 7,680$ sources to fully utilize the machine. To minimize the influence of straggling threads when executing MS-BFS we used three times as many sources for all measurements. All algorithms have to analyze the same set of source vertices that were randomly selected from the graph. For MS-BFS, the sources are processed one 64-vertex batch at a time per CPU core. MS-PBFS can saturate all compute resources with a single 64-vertex batch; it, thus, analyzes one batch at a time. SMS-PBFS analyzes all sources one single source at a time, utilizing all cores.

5.3.1 *Thread Count Scaling*

To ensure that the amount of work is constant in the CPU scaling experiments, we kept the number of sources fixed even when running with fewer cores. The first 15 cores are located on the first CPU socket, 16–30 on the second socket, 31–45 on the third and 46–60 on the fourth. Figure 11 shows that MS-PBFS scales better than MS-BFS even though the latter has no synchronization between the threads. MS-PBFS (sequential) which uses the same optimizations as MS-PBFS but is executed like MS-BFS with one BFS batch per core exhibits the same limited scaling behavior for large graphs. This contradicts the MS-BFS paper's hypothesis that multiple sequential instances always beat a parallel algorithm as no synchronization is required. The explanation for this can be found when analyzing the cache hit rates. With our (S)MS-PBFS algorithms, the different CPU cores share large portions of their working set, and, thus, can take advantage of the sockets' shared last level caches. In contrast, each sequential MS-BFS mostly uses local data structures; only the graph is shared. This diminishes CPU caches' efficiency.

The scalability of around factor 45 for MS-PBFS and factor 35 for SMS-PBFS using 60 threads is comparable to the results reported by Yasui et al. [20] for their parallel single-source BFS. This is a very good result especially as our multi-source algorithm operates at a much higher throughput of 274 GTEPS compared to their best reported result of around 60 GTEPS on a similar machine in the Graph500 benchmark. The close results between the MS-PBFS (one per socket) variant, where all data except for the graph is completely local, and MS-PBFS show that our algorithm is mostly resilient to NUMA effects and is not limited by contention.

When analyzing the performance gains achieved by the

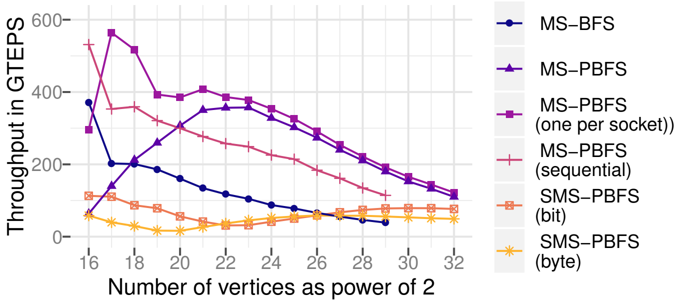


Figure 12: Throughput using 60 cores as graph size increases

additional 60 Hyper-Threads, the difference between multi-source and single-source processing is clearly visible. SMS-PBFS is memory latency-bound and does not saturate the memory bandwidth; thus, it can gain additional performance by having more threads. The multi-source algorithms on the other hand are already mostly memory-bound, and, thus, they do not benefit from the additional threads.

5.3.2 *Graph Size Scaling*

Orthogonal to the thread count scaling experiment, we also measured how the algorithms behave for various graph sizes using Kronecker graphs. We use graph sizes from approximately 2^{16} to 2^{32} vertices and 1 million to 68 billion edges, respectively. As the traversal speed should be independent of the graph size, an ideal result would have constant throughput. Our measured results are shown in Figure 12. MS-BFS as well as the sequential MS-PBFS variant show a continuous decline in performance as the graph size increases. This can be explained with memory bottlenecks, as for larger graph sizes a smaller faction of the graph resides in the CPU cache, and more data has to be fetched from main memory.

In contrast, the parallel BFSs struggle at small graph sizes. Their two biggest problems are contention and that there is only very little work per iteration (on average less than 1 ms runtime). The reason for the contention in very small graphs is the high probability that in the top-down phase multiple threads will try to update the same entry. Furthermore, for small graphs, the constant overheads for task creation, memory allocation, etc., have a relatively high impact on the overall result.

Parallelization is much more important for large graph sizes. Starting at 2^{20} (around 1 million) vertices, MS-PBFS manages to beat the MS-PBFS (sequential) implementation. At this size, MS-PBFS requires 3.27ms for one batch of 64 sources. At larger sizes a decline in performance can be measured again, caused by a reduction in cache hit rates resulting in memory bandwidth bottlenecks. SMS-PBFS maintains its peak performance for a larger range of graph sizes, though at a lower level. As it only operates on a single BFS, it is more bound by memory latency in case of a cache miss than by memory bandwidth. Other BFS approaches [2, 20] also exhibit a similar drop in performance at larger scales. The measurement for MS-BFS and MS-PBFS (sequential) only include graphs up to scale 29, as at larger graph sizes the available one terabyte of memory did not suffice to run 120 instances of the algorithms, demonstrating the severe limitations of MS-BFS in a multi-threaded scenario.

In Table 1 we summarize the algorithms' performance for real world datasets. Additionally, we show the performance when MS-BFS is only limited to processing 64 sources at a time (MS-BFS 64) like MS-PBFS. The results show that in this kind of use case the performance of MS-BFS is very low as it can only utilize one CPU. Overall, our measurements show that even if MS-BFS is given enough sources to utilize all cores, MS-PBFS performs significantly better on large graphs.

We also wanted to compare to the currently fastest parallel single-source BFS by Yasui and fastest parallel multi-source iBFS but did not have access to their implementations. As we could evaluate our algorithms on the same synthetic graphs, we instead compare to their published numbers. For Yasui et al. we compare our SMS-PBFS to their most recent numbers published on the Graph500 ranking. Their results in the Graph500 ranking are based on a CPU that is about 20% faster than ours but from the same CPU generation so they should be comparable. Their result places them 67st overall, 1st single-machine (CPU-only), on the June 2016 ranking and they achieve a throughput of 59.9 GTEPS compared to the 76.7 GTEPS demonstrated by our single-source SMS-PBFS on the same scale 32 graph. For iBFS we use the numbers from their paper, they do not use the default kronecker graph generator settings but test on graphs with larger degrees. We compare MS-PBFS against their algorithm on the KG0 graph where they report their best performance. Using 64 threads, the iBFS CPU implementation reaches 397 GTEPS, and their GPU implementation 729 GTEPS. MS-PBFS reaches 1860 GTEPS on 120 threads showing a significant improvements even when accounting for number of threads.

6. RELATED WORK

The closest work in the area of multi-source BFS algorithms are MS-BFS [18] and the iBFS[14] which is designed for GPUs. Compared to the first algorithm, our parallelized approach using striped labeling significantly improves the performance, and reduces the memory requirements. Furthermore, MS-PBFS enables the use of multi-source BFS in a wider setting by providing full performance also with a limited number of sources. iBFS describes a parallelized approach for GPUs which uses a sparse joint frontier queue (JFQ) containing the active vertices for a iteration. By using special GPU voting instructions, it manages to avoid queue contention on the GPU. However, those instructions don't have equivalents on mainstream CPU architectures. Consequently, the CPU adaption of their algorithm exhibits significantly lower performance than ours.

The work on parallel single-source algorithms primarily focuses on how to reduce the cost of insertion into the *frontier* and *next* queues. Existing approaches span from using multi-socket-optimized queues like FastForward [10], to batch insertions and deletions [2], as well as to having a single queue per NUMA node as it is used by the parallel Yasui BFS [20]. Yet, all of these approaches have in common that they share a single insertion point either at the global level or per NUMA node. Even if organized at NUMA socket granularity there is potentially a lot of contention, and the trend of having more cores per CPU does not work in such approaches' favor. The work of Chhugani et al. [8] which also focuses on dynamic load balancing has similar limitations as it only focuses on distributing work inside each NUMA socket. Our analysis shows that while this may be sufficient for sparse queue-based algorithms, it would not provide scalability in array-based algorithms.

7. CONCLUSION

In our work we presented the MS-PBFS and SMS-PBFS algorithms that improve on the state-of-the art BFS algorithms in several dimensions.

MS-PBFS is a parallel multi-source breadth-first search that builds on MS-BFS's principles of sharing redundant traversals in concurrent BFSs in the same graph. In contrast to MS-BFS, our novel algorithm provides CPU scalability even for a limited number of source vertices, fully utilizing large NUMA systems with many CPU cores, while consuming significantly less memory, and providing better single-threaded performance. Our parallelization and NUMA optimizations come at minimal runtime costs so that no separate algorithms are necessary for sequential and parallel processing, neither for NUMA and non-NUMA systems.

SMS-PBFS is a parallel single-source BFS that builds on the ideas of MS-PBFS. Compared to existing state-of-the-art single-source BFSs, our proposed SMS-PBFS algorithm provides comparable scalability at much higher absolute performance. Unlike other BFS algorithms, SMS-PBFS has a simple algorithmic structure, requiring very few atomic instructions and no complex lock or queue implementations. Our novel striped vertex labeling allows more coarse-grained task sizes while limiting the skew between task runtimes. Striped vertex labeling can also be used to improve the performance of other BFS algorithms.

8. ACKNOWLEDGMENTS

Manuel Then is a recipient of the Oracle External Research Fellowship.

9. REFERENCES

[1] The Graph 500 Benchmark. http://www.graph500.org/specifications. Accessed: 2016-09-09.

[2] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *Proc. of the 22nd IEEE and ACM Supercomputing Conference (SC10)*, SC '10, pages 1–11. IEEE, 2010.

[3] L. A. N. Amaral, A. Scala, M. Barthélémy, and H. E. Stanley. Classes of small-world networks. *PNAS*, 97(21), 2000.

[4] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Graph Partitioning and Graph Clustering*, volume 588 of *Contemporary Mathematics*. American Mathematical Society, 2013.

[5] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[6] S. Beamer, K. Asanovic, and D. A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[7] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW '04*, pages 595–601, 2004.

[8] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 378–389, May 2012.

[9] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 619–630. ACM, 2015.

[10] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 43–52. ACM, 2008.

[11] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. WTF: The Who to Follow Service at Twitter. In *WWW '13*, pages 505–514, 2013.

[12] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 78–88, Oct 2011.

[13] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.*, 11:985–1042, Mar. 2010.

[14] H. Liu, H. H. Huang, and Y. Hu. iBFS: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 403–416. ACM, 2016.

[15] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[16] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146. ACM, 2010.

[17] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 445–456, Oct 2015.

[18] M. Then, M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The more the merrier: Efficient multi-source graph traversal. *Proceedings of the VLDB Endowment*, 8(4), 2014.

[19] Y. Yasui and K. Fujisawa. Fast and scalable NUMA-based thread parallel breadth-first search. In *High Performance Computing & Simulation (HPCS)*, pages 377–385. IEEE, 2015.

[20] Y. Yasui, K. Fujisawa, and Y. Sato. Fast and energy-efficient breadth-first search on a single NUMA system. In *Proceedings of the 29th International Conference on Supercomputing*, ISC 2014, pages 365–381. Springer-Verlag New York, Inc., 2014.

[21] 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 *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2. USENIX Association, 2012.

GraphCache: A Caching System for Graph Queries

Jing Wang
School of Computing Science
University of Glasgow, UK
j.wang.3@research.gla.ac.uk

Nikos Ntarmos
School of Computing Science
University of Glasgow, UK
nikos.ntarmos@glasgow.ac.uk

Peter Triantafillou
School of Computing Science
University of Glasgow, UK
peter.triantafillou@glasgow.ac.uk

ABSTRACT

Graph query processing is essential for graph analytics, but can be very time-consuming as it entails the NP-Complete problem of subgraph isomorphism. Traditionally, caching plays a key role in expediting query processing. We thus put forth GraphCache (GC), the first full-fledged caching system for general subgraph/supergraph queries. We contribute the overall system architecture and implementation of GC. We study a number of novel graph cache replacement policies and show that different policies win over different graph datasets and/or queries; we therefore contribute a novel hybrid graph replacement policy that is always the best or near-best performer. Moreover, we discover the related problem of cache pollution and propose a novel cache admission control mechanism to avoid cache pollution. Furthermore, we show that GC can be used as a front end, complementing any graph query processing method as a pluggable component. Currently, GC comes bundled with 3 top-performing filter-then-verify (FTV) subgraph query methods and 3 well-established direct subgraph-isomorphism (SI) algorithms – representing different categories of graph query processing research. Finally, we contribute a comprehensive performance evaluation of GC. We employ more than 6 million queries, generated using different workload generators, and executed against both real-world and synthetic graph datasets of different characteristics, quantifying the benefits and overheads, emphasizing the non-trivial lessons learned.

CCS Concepts

•**Information systems** → **Database query processing;**

Keywords

Graph analytics; Graph queries, Caching system

1. INTRODUCTION

Graph datasets are proliferating nowadays, due to their ability to capture and allow for the analysis of complex relations among objects, by modelling entities with nodes and their relations/interactions with edges. Graphs have thus been used, with great success, in a wide variety of application areas, from chemical and bioinformatics datasets to social networks. Central to graph analytics is the ability to locate patterns in dataset graphs. Informally, given a query (pattern) graph g, the system is called to return the set of dataset graphs that contain g (subgraph query) or are contained in g (supergraph query), aptly named the *answer set* of g. Unfortunately, these operations can be very costly, as they entail the NP-Complete problem of subgraph isomorphism[5] and even the popular algorithms [4, 18, 31] are known to be computationally expensive. To this end, the research community has contributed a number of innovative solutions over the last few years. A large number of these follow the "filter-then-verify" (FTV) paradigm: dataset graphs are indexed so as to allow for the exclusion (filtering) of a number of those that are definitely not in the query's answer set; the remaining graphs, called the *candidate set* of g, need then to undergo testing (verification) for subgraph isomorphism (abbreviated as *sub-iso* or *SI* in the rest of this work). However, recently extensive evaluations of FTV methods [9, 12] show significant performance limitations.

Although FTV solutions can produce candidate sets that are much smaller than the original dataset, they still end up executing unnecessary sub-iso tests: in the simplest of cases, if the same query is submitted twice to the system, it will also be sub-iso tested twice against its candidate set. Furthermore, a key observation we can make is that in many real-world applications, graph queries submitted in the past bear subgraph or supergraph relations with future queries. These relationships arise naturally. Queries against a biochemical dataset range from queries for simple molecules and aminoacids, all the way to queries for proteins of multi-cell organisms. In exploratory smart-city analytics, queries referring to road networks may pertain to neighbourhoods, towns, metro areas, etc. In social networking queries, exploratory queries may start off broad (e.g., all people in a geographic location) and become increasingly narrower (e.g., by homing in on specific demographics). In time-series graph analytics, queries are typically associated with time intervals, which contain (or are contained within) other intervals.

Based on these observations, we proposed in [34] a new graph query processing method, in which queries (and their answers) are indexed and used to expedite future query processing with FTV methods. Underpinned by our method in [34], this work presents a novel full-fledged caching system, where any general subgraph/supergraph query method in

Series ISSN: 2367-2005 10.5441/002/edbt.2017.03

the literature could be plugged in, and overall contributes:

- GraphCache (GC), a full-fledged caching system for sub-/supergraph queries, with detailed discussions of design issues, its architecture and implementation, dealing with resource management (memory and threads) and dynamic management of the cache index;
- A fresh perspective to expedite state-of-the-art solutions for the general subgraph isomorphism problems (SI methods) by GC (in addition to FTV methods);
- A semantic graph cache which harness sub/supergraph cache hits, extending the traditional exact-match-only hit and leading to significant speedup for GC;
- A number of graph cache replacement strategies with different trade-offs, including a novel hybrid graph cache replacement policy with performance always better or on par with the best alternative;
- A novel cache admission control mechanism enhancing the performance gains of GC;
- Comprehensive evaluations (with millions of queries) utilising well-established FTV and SI methods, against real-world and synthetic datasets with different characteristics and different workload generators, quantifying benefits/overheads and uncovering key insights.

To the best of our knowledge, GC is the first caching system in the literature for general subgraph/supergraph queries.

2. RELATED WORK

Subgraph/supergraph queries entail the subgraph isomorphism problem, which has two versions. The decision problem answers Y/N as to whether the query is contained in each graph in the dataset. The matching problem locates all occurrences of the query graph within a large graph (or a dataset of graphs). For both the decision and matching problems, the brute-force approach is to execute sub-iso tests of the query against all dataset graphs. However, sub-iso tests are costly, being NP-Complete[5]. Several heuristic algorithms have been proposed over the years. [9] provides an insightful presentation and comparison of several such (SI) algorithms (which could be integrated within GC).

SI algorithms deteriorate when the dataset is comprised of a large number of graphs, as each graph has to be tested. Thus appeared the "filter-then-verify" (FTV) paradigm. FTV methods try to reduce the set of graphs against which to run the sub-iso test, by filtering out graphs which definitely do not belong to the query answer set. At the heart of these methods lies an index on the dataset graphs. Briefly, dataset graphs are decomposed into features (i.e., paths, trees, cycles or arbitrary subgraphs), which are then recorded in an indexing structure (e.g., trie [2, 6], hash-based bitmap [14], etc.). Query processing then proceeds in two stages. First, in the *filtering stage*, the query graph g is decomposed to its features, which are then used to retrieve from the dataset index the IDs of those graphs containing all of them; the result is a subset of the dataset graphs, named the *candidate set* of g. Then, in the *verification stage*, g undergoes sub-iso testing against each graph in the *candidate set.*

Similarly, [29] presents a solution for subgraph queries against historical (i.e., snapshotted) graphs – a variation of typical graph queries where snapshots can be viewed as different graphs; the main focus of this work is on reconstructing minimal snapshots around candidate matching nodes, by using a set of indices allowing for the retrieval of nodes with specific labels/neighbourhoods at given time points. [9] presents an insightful performance evaluation and [12] provides a systematic performance and scalability study of subgraph FTV methods. Though we are not aware of similar in-depth studies on supergraph FTV solutions, [36] provides a concise overview for studies published prior to 2013 and recently [20] proposes an efficient solution for supergraph queries. GC is capable of expediting query processing for both subgraph and supergraph FTV methods.

The community has also looked into subgraph queries against a single, very large graph (consisting of possibly billions of nodes). [16] and [30] employ scale-out architectures and large memory clusters with massive parallelism respectively. [8] and [28] provide a centralised solution to the same problem, via advanced pruning approaches addressing the matching order issues faced by most other SI algorithms. GC does not target such use cases for the time being and extending our system to queries against a single massive graph or distributed operation is left for future work.

Caching of query results has long been a mainstay in data management systems, from filesystem block caching to web proxy caching and the cache of query result sets in relational databases. In the realm of graph-structured queries, however, little work has been done. For XML datasets, views have been used to accelerate path/tree queries [1, 19, 21]; Besides, [17] firstly proposed the MCR (maximally contained rewriting) approach for tree pattern queries and [33] revisited it by providing alternatives; both exhibit false negatives for the query answer. Our GC does not produce any false negative or false positive (formal proof of correctness in [34]). Also, GC is capable of dealing with much more complex graph-structured queries, which entail the NP-Complete problem of subgraph isomorphism.

More recently, caching has also been utilized to optimize SPARQL query processing for RDF graphs. [22] introduced the first SPARQL cache, where a relational DB was employed to store the metadata. [27] contributed a cache for SPARQL queries based on a novel canonical labelling scheme (to identify cache hits) and on a popular dynamic programming planner [23]. Similar to GC, query optimization in [27] does not require any a priori knowledge on datasets/-workloads and is workload adaptive. However, like XML queries, SPARQL queries are less expressive than general graph queries and thus less challenging [13, 30]; SPARQL query processing consists of solving the subgraph homomorphism problem, which is different from the subgraph isomorphism problem, as the former drops the injective property of the latter. Moreover, GC discovers subgraph, supergraph, and exact-match relationships between a new query and the queries in the cache, something that the canonical labelling scheme in [27] fails to achieve. SPARQL query processing also aims at optimizing join execution plans [7] (based on join selectivity estimator statistics and related cost functions), and the cache in [27] is focusing on this goal, whereas GC aims to avoid/reduce costs associated with executing SI heuristics whose execution time can be highly unpredictable and much higher. As such, the overall rationale of GC and the way cache contents are exploited differs from that in [27] and in related SPARQL result caching solutions.

Finally, [15] presents a cache for historical queries against a large social graph, in which each query is centered around a node in the social graph, and where the aim is to avoid maintaining/reconstructing complete snapshots of the social graph but to instead use a set of static "views" (snapshots of neighborhoods of nodes) to rewrite incoming queries. [15] does not deal with subgraph/supergraph queries per se; rather, the nature of the queries means that containment can be decided by just measuring the distance of the central query node to the centre of each view. Moreover, [15] does not deal with central issues of a cache system (cache replacement, admission control, overall architecture/design, etc.).

3. DESIGN ISSUES AND GOALS

GraphCache is implemented for undirected labelled graphs, as is typical in the literature (e.g., [2, 10, 14]). For simplicity, we assume that only vertices have labels; all our results straightforwardly generalize to directed graphs and/or graphs with edge labels.

Formally, a labelled graph $G = (V, E, l)$ consists of a set of vertices V and edges $E = \{(u, v), u, v \in V\}$, and a function $l : V \rightarrow U$, where U is the domain of labels. A graph $G_i = (V_i, E_i, l_i)$ is subgraph-isomorphic to a graph $G_j = (V_j, E_j, l_j)$, by abuse of notation denoted by $G_i \subseteq G_j$, when there exists an injection $\phi : V_i \rightarrow V_j$, such that $\forall (u, v) \in E_i, u, v \in V_i, \Rightarrow (\phi(u), \phi(v)) \in E_j$ and $\forall u \in V_i, l_i(u) = l_j(\phi(u))$. Informally, there is a subgraph isomorphism $G_i \subseteq G_j$ if G_j contains a subgraph that is isomorphic to G_i, and we say that G_i is a subgraph of (contained in) G_j, or that G_j is a supergraph of (contains) G_i denoted by $G_j \supseteq G_i$. As is common in the relevant literature, we focus on non-induced subgraph isomorphism. Last, the subgraph (supergraph) querying problem entails a set $D = \{G_1, \dots, G_n\}$ containing n graphs, and a query graph g, and determines all graphs $G_i \in D$ such that $g \subseteq G_i$ ($g \supseteq G_i$, respectively).

In designing GraphCache, we identified a set of design issues and goals, pertaining to the characteristics of (i) the query workloads, (ii) the underlying graph datasets, and (iii) the algorithmic and system context within which GC will operate (e.g., categories of research methods GC will complement). Overall, GC is intended to expedite graph queries whatever the algorithm of choice may be and across a wide variety of query workloads and graph datasets.

Query Workloads. As with any caching system, the assumption is that previous queries can help expedite future queries. This is reasonable, given the example applications mentioned in §1. Most works [2, 6, 9, 14, 35] test algorithms for queries directly generated from dataset graphs. Though this is of particular interest, workloads should also include queries that are not guaranteed to have any answer. Furthermore, in general, of particular interest to any caching system is the probability distribution of possible queries. For GC this in effect refers to the popularity of query graphs or of regions of the dataset graphs. GC should thus be able to deal effectively with various skewness levels of this distribution (e.g., from uniform to highly skewed Zipf distributions). Finally, a practical problem emerges: workloads must contain a large number of queries so as to obtain reliable results on the performance of any method but subgraph isomorphism is NP-Complete. This leads to queries with possibly very long execution times, regardless of the heuristic used, making the experiments very time consuming. Nevertheless, we utilized well over 6 million queries for our performance evaluation.

Graph Datasets. Fortunately there exist a number of real-world graph datasets commonly used in related research. These help concretize the effects of any solution on real-world data and allow direct comparison of methods and result repeatability. For this reason we will report evaluations conducted over three popular graph datasets: AIDS[24], PDBS[11], and PCM[32]. However, it is worth creating additional synthetic datasets so to perform evaluations under characteristics unseen in the real-world datasets. Specifically, we created a synthetic dataset presenting interesting characteristics regarding the number, size and node degrees of graphs in the dataset. Interestingly, with respect to dataset with graphs having a high average node degree, we found that GC needs special mechanisms without which its performance benefits degrade.

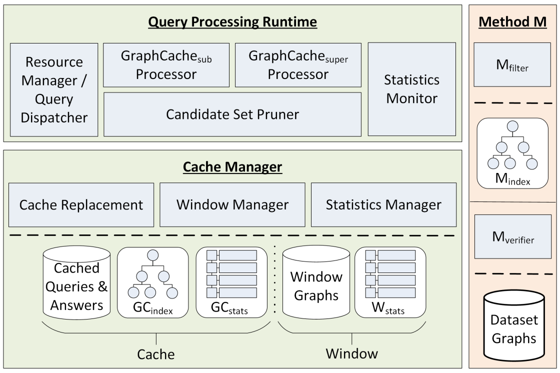


Figure 1: GraphCache System Architecture

Algorithmic Context. GraphCache is intended to be a general-purpose front-end for graph query processing. GC entails a query indexing strategy that, as explained in [34], can accommodate both subgraph and supergraph queries. In addition, the design of GraphCache must be able to accommodate both FTV methods and SI algorithms; its current implementation comes bundled with well-established FTV methods and SI algorithms. In fact, any such algorithm is viewed as a pluggable component into the architecture, allowing any future algorithm to be incorporated.

4. SYSTEM ARCHITECTURE

GraphCache is designed from the ground up as a scalable semantic cache for subgraph/supergraph queries, capable of expediting any SI or FTV method (henceforth denoted *Method M*). Figure 1 shows its main architectural components, comprising three major subsystems: Method M, Query Processing Runtime, and Cache Manager. The last two are internal subsystems of GC; the first is the method that GC is called to expedite and hence external to GC.

The Method M subsystem includes, at a minimum, the base graph dataset and a sub-iso test implementation, denoted M$_{verifier}$. Additionally, if M is a FTV method, then it also features its index, denoted M$_{index}$, and a filtering component, M$_{filter}$. The index is built in a pre-processing step, by using Method M's indexing component (not shown in Figure 1 for simplicity). When GC is not used, sub-/supergraph query processing proceeds by first using M$_{index}$ through M$_{filter}$ to prune away dataset graphs definitely not containing (or contained in) the query, thus forming its candidate set, M$_{CS}$. Then M$_{verifier}$ executes a sub-iso test against all graphs in M$_{CS}$, reading their structure directly from the graph dataset store. For SI methods, M$_{CS}$ contains all graphs in dataset.

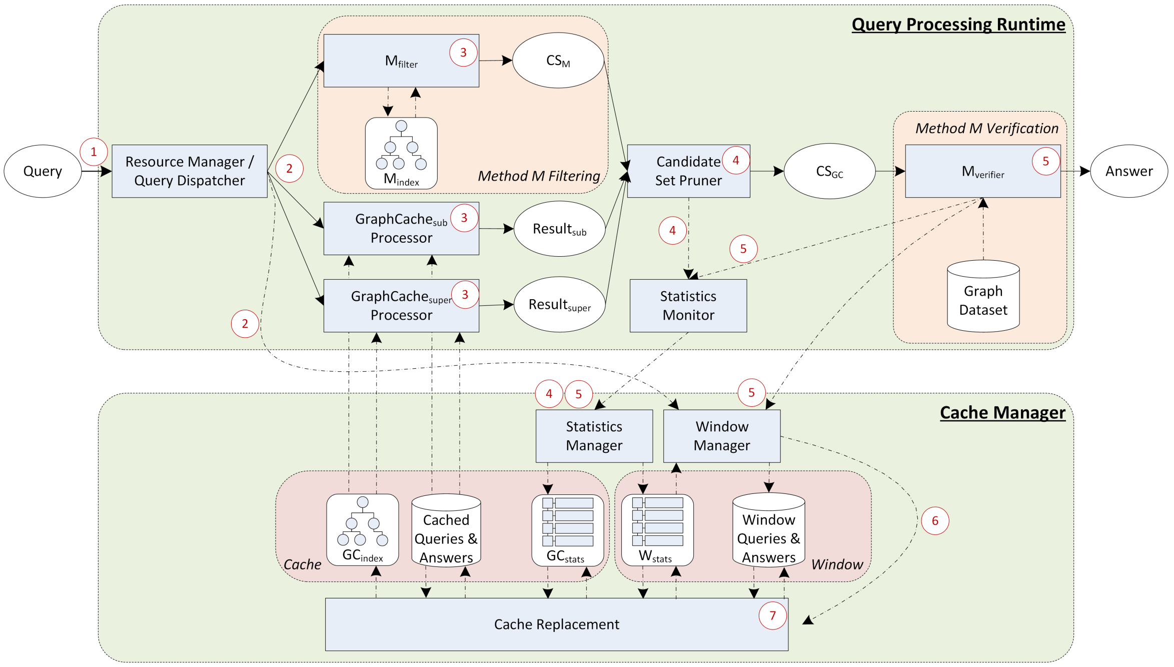


Figure 2: GraphCache System Data and Control Flow

Within GC, the Query Processing Runtime is responsible for the execution of queries and the monitoring of key operational metrics. It comprises: a resource/thread manager dispatching queries to the various filtering/verification modules, the internal subgraph/supergraph query processors, the logic for GC's candidate set pruning, and a statistics monitor. These components communicate with Method M and the Cache Manager via well-defined APIs.

In turn, the Cache Manager deals with the management of data and metadata stored in the cache. It comprises the cache replacement mechanisms, a Window Manager responsible for cache admission control and maintenance of the cache contents, a Statistics Manager responsible for metadata pertaining to past or current queries, as well as the stores for all GC-related data including cached queries and their answer sets, currently executing (not cached) queries, and metadata/statistics for both past and current queries.

Figure 2 depicts the flow of control and data in GC during processing of a query. The query first arrives at the Resource Manager (1) and is then dispatched to M_{filter} and GC's filtering processors in parallel (2). At the same time, a copy of the query is added to the set of currently processed queries, called the Window (discussed shortly). The filtering components use their respective indexes to produce intermediate candidate sets (3). More specifically, M_{filter} uses M_{index}, while the two GraphCache processors use GC_{index}, the set of cached graph queries and their answer sets. The results of this stage are then fed to the Candidate Set Pruner which produces the final candidate set GC_{CS} (4); at the same time, statistics regarding GC_{CS} and the contribution of cached graphs are gathered by the Statistics Monitor and forwarded to the Statistics Manager. The final candidate set then undergoes sub-iso testing using $\mathrm{M}_{verifier}$(5); metadata pertaining to the verification time are also gathered by the Statistics Monitor and sent to the Statistics Manager. When the Window is full, the Window Manager selects the set of current queries to be considered for admission in the cache (6) and invokes the cache replacement algorithm (7); i.e., updates to the Cache are batched through the Window.

5. QUERY PROCESSING

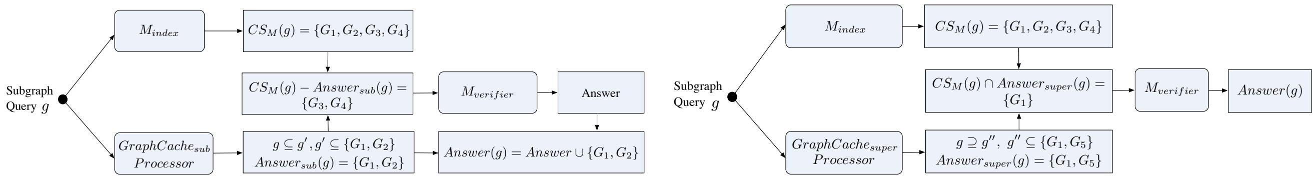
This section discusses the design and implementation of GraphCache's Query Processing subsystem, responsible for the execution of queries and the monitoring of key operational metrics. For the sake of clarity we first describe GC's operation when caching subgraph queries; we shall then discuss how GC can be used for supergraph queries as well.

5.1 Candidate Set Pruning

This subsection overviews the essence of [34]. For more details and formal proofs of correctness, please refer to [34]. Initially, if Method M is a FTV method, its indexing subsystem is used to build its graph dataset index as per usual. The GraphCache's data stores are initially all empty and are then populated as queries arrive and are processed. When a query g arrives at the system, M_{filter} is used to produce a first candidate set. Concurrently, GraphCache checks whether the query graph is a subgraph or supergraph of previous query graphs, through its GC_{sub}/GC_{super} Processors.

GraphCache$_{sub}$ Processor. The GraphCache$_{sub}$ Processor is responsible for identifying when a new query g is a subgraph of a previous query g'. When g' was executed , GC indexed g''s features in GC_{index} and stored its result set and relevant statistics in the cache data stores.

Figure 3(a) depicts an example flowchart for this case. The new query g is processed through M_{filter}, producing candidate set $CS_M(g)$ (with four graphs $\{G_1, G_2, G_3, G_4\}$). Similarly, g is processed by the GC_{sub} Processor, determining that there exists a previous query g', such that $g \subseteq g'$. GC then retrieves g''s cached answer set, $\{G_1,\ G_2\}$. Now, consider graph $G_1 \in CS_M(g)$. Since $g \subseteq g'$ and from the answer set of g' we know that $g' \subseteq G_1$, it necessarily follows that $g \subseteq G_1$ (and, similarly, $g \subseteq G_2$). Therefore, we


Subgraph Query g
M_{index}
$CS_M(g) = \{G_1, G_2, G_3, G_4\}$
$CS_M(g) - Answer_{sub}(g) = \{G_3, G_4\}$
$M_{verifier}$
Answer
$GraphCache_{sub}$ Processor
$g \subseteq g', g' \subseteq \{G_1, G_2\}$
$Answer_{sub}(g) = \{G_1, G_2\}$
$Answer(g) = Answer \cup \{G_1, G_2\}$
Subgraph Query g
M_{index}
$CS_M(g) = \{G_1, G_2, G_3, G_4\}$
$CS_M(g) \cap Answer_{super}(g) = \{G_1\}$
$M_{verifier}$
$Answer(g)$
$GraphCache_{super}$ Processor
$g \supseteq g'', g'' \subseteq \{G_1, G_5\}$
$Answer_{super}(g) = \{G_1, G_5\}$

(a) Subgraph Case

(b) Supergraph Case

Figure 3: GraphCache Processing of a Subgraph Query g

can safely remove G_1 and G_2 from $CS_M(g)$ and add them directly to the final answer set. In the general case, g may be a subgraph of multiple previous query graphs g'_i. Then, the set of graphs that need be sub-iso tested is given by:

$$CS_{GC_{sub}}(g) = CS_M(g) \ \setminus \bigcup_{g'_i \in Result_{sub}(g)} Answer(g'_i) \quad (1)$$

where Result$_{sub}(g)$ contains all query graphs currently in GC$_{index}$ of which g is a subgraph.

GraphCache$_{super}$ Processor. In turn, the GC$_{super}$ Processor is responsible for identifying when a new query g is a supergraph of a previous query g''. Figure 3(b) depicts an example flowchart for this case. Again, Method M produces its candidate set, $CS_M(g)$ (e.g., $\{G_1, G_2, G_3, G_4\}$). GC$_{super}$ then determines that there exists a previous query graph g'' such that $g'' \subseteq g$ and whose cached answer set is $\{G_1, G_5\}$. The reasoning then proceeds as follows. Consider graph $G_2 \in CS_M(g)$. We know from the cached answer set above that G_2 is not in the answer set of g''. Since $g'' \subseteq g$, if $g \subseteq G_2$ were to be true then it should also hold that $g'' \subseteq G_2$; i.e., the answer set of g'' would contain G_2, which is a contradiction. Therefore, it is safe to conclude that $g \nsubseteq G_2$ and thus G_2 can be removed from $CS_M(g)$. In the general case, g may be a supergraph of multiple previous query graphs g''_j. Then, the set of graphs tested for sub-iso by GC is:

$$CS_{GC_{super}}(g) = CS_M(g) \ \cap \bigcap_{g''_j \in Result_{super}(g)} Answer(g''_j) \quad (2)$$

where Result$_{super}(g)$ contains all query graphs currently contained in GC$_{index}$ of which g is a supergraph.

Putting It All Together. The Candidate Set Pruner collects CS$_M$ and the results of the above two Processors; it then first applies equation (1) on CS$_M$, then applies (2) on the result of the previous operation. The end result is a reduced candidate set, which is then sub-iso tested by M$_{verifier}$.

Two Special Cases. Additionally, there are two cases that warrant attention since they yield the greatest possible gains.

First, note that GC can easily recognize the case where a new query, g, is isomorphic to a previous cached query. For connected query graphs, this holds when $\exists g' \in GC_{index}$ such that $g \subseteq g'$ or $g \supseteq g'$, and g and g' have the same number of nodes and edges. Thus, GC can return the cached result of g' directly and completely avoid any further processing.

Second, consider that $\exists g' \in Result_{super}(g)$ (i.e., $g' \subseteq g$) and $Answer(g') = \emptyset$; then GC can directly return with an empty result set. The reason is that if there were a dataset graph g'' such that $g \subseteq g''$, since $g' \subseteq g$ we would conclude that $g' \subseteq g''$, which implies that $g'' \in Answer(g')$, contradicting the fact that $Answer(g') = \emptyset$; thus, no such graph g'' can exist and the final result set is necessarily empty.

Supergraph Query Processing. As mentioned earlier, GC can expedite both subgraph *and* supergraph query processing. In the latter case, the filtering components of GC remain unchanged, but the handling of the return answer sets is the exact inverse of what happens for subgraph queries. Briefly, given a supergraph query processing Method M and a supergraph query g, the union of the answer sets of graphs in $Result_{super}(g)$ are removed from $CS_M(g)$ and added to $Answer_{GC_{super}}(g)$, and the graphs not appearing in the intersection of the answer sets of graphs in $Result_{sub}(g)$ are completely subtracted from $CS_M(g)$. Also, the first special case still holds, but for the second special case processing terminates when $\exists g' \in Result_{sub}(g)$ such that $Answer(g') = \emptyset$.

5.2 Statistics Monitoring

The final component of this subsystem is the Statistics Monitor. This is a lightweight layer, implemented as a wrapper library allowing components of this subsystem to record various statistics (see §6.1) and to communicate them to the Statistics Manager component of the Cache Manager subsystem. Currently the following quantities are monitored:

- Static query metrics such as the number of nodes, edges and distinct labels in the query.
- Total filtering and verification time of the query when first executed.
- Break-down of total filtering times of the query to the three filtering components.
- Number of times the query was matched by either GC Processors and number of special-case matches.
- Most recent time a cached query was hit, expressed as the serial no. of last benefited query.
- Total reduction in the candidate set size of new queries. This statistic is easily monitored, as the Candidate Set Pruner knows exactly which graphs from the answer set of each matched cached query where removed from the candidate set of any given new query (through application of equations (1) and (2)).
- Total time saving due to the cached query. This statistic is computed as the sum of the estimated costs of all sub-iso tests alleviated, as mentioned above. The estimation of the individual sub-iso test time $c(g, G)$ for a query graph g against a dataset graph G, is performed using the formula[34]: $c(g, G) = \frac{N \times N!}{L^{n+1} \times (N-n)!}$, where L is the number of distinct labels, n the number of nodes in g, and $N \geq n$ the number of nodes in G.

6. CACHE MANAGEMENT

GC's Cache Manager subsystem, running in parallel with the Query Processing Runtime subsystem, deals with the management of the data and metadata stored in the cache.

We first discuss the various data stores handled by this subsystem, then dive into the design of its various components.

6.1 Data Layer

GraphCache's Cache Manger maintains a number of complementary data stores, conceptually bundled together into two groups: the Cache stores and the Window stores.

The Cache stores include three components: First, a component storing copies of cached queries (i.e., the actual graph submitted as a query to GC) alongside their result sets (i.e., the sets of dataset graph IDs containing (for subgraph queries) or being contained in (for supergraph-queries) the query graph). This component is implemented as an in-memory hash table, loaded from disk on startup and written back to disk on shutdown of the Cache Manager subsystem. In said hash table, the serial number of the query is used as the key and the query graph and result set as the value. At startup, an upper limit is set on the size of this hash table (expressed in number of records); the Cache is deemed full when this upper limit is reached. Second, a combined subgraph/supergraph index, indexing the aforementioned query graphs to expedite subgraph/supergraph matching of future queries against past queries. We have loosely based our query index design on the GraphGrepSX subgraph query index[2], augmented with additional metadata to allow for the processing of supergraph queries. This index is loaded on startup and written back on shutdown of the Cache Manager subsystem. Our index design allows us to have a single index for both subgraph and supergraph queries, thus providing for lower disk space and I/O overhead, and a memory footprint low enough to allow for the index to be easily resident in main memory throughout the lifetime of the Cache Manager process. Third, a component storing statistics for each cached query, implemented as an in-memory key-value store, loaded from disk on startup and written back on shutdown of GC. The query serial number is again used as the key, pointing to a variable size array of columns, sorted by column name. Columns in this store include, but are not limited to: static query such as the number of nodes, edges and distinct labels in the query; total filtering and verification time of the query when first executed; count of times the query was matched by either of the $\mathrm{GC}_{sub}/\mathrm{GC}_{super}$ Processors plus number of optimal matches (see §5.1); last (most recent) time a query contributes, expressed as the serial number of the benefited query; total contribution of the cached query in reducing the candidate sets and processing times of future queries, expressed as the number of dataset graphs removed from the candidate set of queries due to their being in the cached query's answer set and the cumulative sub-iso test time alleviated; etc.

On the other hand, the Window stores include two components: First, a component storing new graph queries and their result sets, implemented in the same manner as the first component of the Cache stores above. An upper limit on the size of this store is also configured at startup; the Window is deemed full when said limit is reached. Second, a component storing statistics for each query in the previous component, also implemented as an in-memory key-value store like the statistics component of the Cache stores. In this case, the statistics include only static information regarding the new queries, including the number of nodes, edges and distinct labels in the query, as well as the total filtering and verification time of the query.New queries are sent to the Window Manager directly from the Query Dispatcher to be added to the appropriate store, while their answer sets are added at the end of their processing.

All updates to the query statistics stores are performed through the Statistics Manager using values supplied by the Statistics Monitor. The Statistics Manager is currently implemented as a lightweight wrapper library, encapsulating accesses to the statistics stores. The design of this subsystem has explicitly been abstract enough to allow for an easy replacement of the data stores with other in-memory, on-disk or even remote/distributed stores without requiring changes to the rest of our code. The Statistics Manager exposes an interface akin to that of contemporary key-value stores; i.e., it stores triplets of the form {key, column name, column value}, accessible either by key (returns a "row" with all triplets with the given key), or by column name alone (returns a "column" with all triplets with the given column name), or by key and column name (returns a single triplet).

6.2 Window Manager with Admission Control

The Window Manager, implemented as a separate thread, is the brain of the Cache Manager subsystem. It keeps track of the queries in the current Window and invokes the Cache Admission Control algorithm to decide whether each new query should be considered as a candidate for addition to the cache. It also executes the Cache Replacement algorithms when the Window is full, and rebuilds GC_{index} to reflect any changes in the cached queries store. In the latter case, the Window Manager first computes the new contents of the cache (by replacing evicted queries with admitted Window queries) and invokes the indexing mechanism; queries arriving at the system while this procedure is taking place, continue being served by the old index and update the old statistics. Once the re-indexing is over, the new cache contents and index are swapped in place of the old ones, and any statistics entries corresponding to evicted queries are removed lazily from the statistics store. The driving force behind this design was the fact that, much like all index-based graph-matching methods, our current version of GC_{index} does not support dynamic concurrent updates. Nevertheless, our design allows for low-latency/high-throughput processing of new queries, even while the index is rebuilt, and incurs minimal locking overhead (i.e., only for the swapping of old and new cache contents/index structures, actually implemented as simple in-memory reference (pointer) swaps), trading off some possible cache hits against window queries.

Cache Admission Control. While experimenting with different workloads and datasets we observed that often the performance of GraphCache would be lower than expected; that is, although GraphCache benefited the majority of queries, the overall speedup achieved was very low (close to 1). The reason behind this proved to be that the cache was *polluted*, storing and improving the performance primarily of inexpensive graph queries. To alleviate this situation, we make the natural conjecture that past expensive (time-wise) queries are more likely to benefit later coming expensive queries as they will help in alleviating more expensive sub-iso tests (and vice-versa for inexpensive queries). We therefore propose a novel admission control mechanism, part of the Window Manager component, which optimises the graph cache by preventing inexpensive queries from being added to the cache. To quantify the expensiveness of a

Table 1: Running Example: Cached Query Statistics

SerialNo / Query ID	Last Hit	Number of Hits	CS$_M$ Reduction	SI Cost Reduction
11	91	23	170	2600
13	51	32	80	1200
37	69	26	376	780
53	78	13	210	360
82	90	5	120	150
91	95	4	10	270

query graph, we use the ratio of its verification time over its filtering time. Each executed query is thus assigned an "expensiveness" score and only queries with such a score above a threshold are considered as candidates for entering the graph cache (a threshold value of 0 disables this component). To compute said threshold, our mechanism examines the queries in the first few *windows* and computes an expensiveness value which would result in a predefined percentage of queries being classified as expensive. We have also experimented with more dynamic approaches (e.g., greedily adapting the threshold using an exponential back-off approach until the achieved time speedup reaches a local maximum); without loss of generality and due to lack of space, we omit further discussion of these techniques. The reasoning behind the above lies in the fact that, given a graph query processing framework, the filtering time is relatively constant across queries, in contrast to the dramatic variance of verification times. Moreover, the verification stage is known to dominate the query time[9, 12], and the larger the verification time the more overwhelming this dominance. Thus, the above mechanism is a simple yet effective technique to guarantee that more complex queries are prioritised.

6.3 Cache Replacement Policies

[34] used a specific graph replacement policy (PINC). We have developed and tested a number of new different cache replacement strategies (POP, PIN and HD), each offering different trade-offs and performance characteristics for different datasets and query workloads. We describe the various strategies here and report on their relative performance in §7. In all cases, the replacement strategies access query statistics through the Statistics Manager's key-value store interface, and return the IDs of queries to be cached out. In order to compute this set, queries are assigned a "utility" value and those with the lowest such values are cached out.

Below we present all cache replacement algorithms considered in this work. We use Table 1, presenting a snapshot of GC$_{stats}$ for a number of hypothetical cached queries, as our running example. In all cases, assume that the replacement algorithm is invoked at time point 99 (i.e., right after the query with serial number 99 was executed) and needs to remove two entries from the cache, thus has to find the two entries with the lowest utility value.

Least Recently Used (LRU). LRU discards the least recently used items from the cache. The utility of each cached graph is its last hit time, i.e., the serial no. of the last query that is expedited by said cached graph. In our running example, cached queries with serial number 13 and 37 would be cached out. LRU is a simple and very popular policy in several traditional caches. However, it builds on the assumption of temporal locality of reference and thus fails to identify cases of queries which have contributed huge savings to query processing although not having been used in a while. In our example, we can see that query 13 has contributed the most times, but still is evicted.

Popularity-based Ranking (POP). Ideally we would prefer a replacement policy that would take into account the *popularity* of queries. This leads to the second policy considered here: POP (short for Popularity-based Ranking). This policy assigns each cached graph a utility value equal to H/A, where H is the number of times a query has contributed and A is its age in the cache, computed as the number of processed queries since the said graph enters cache; this function manages to combine query popularity and age. In our example, this policy would evict queries 11 and 53.

POP + Number of Sub-Iso Tests (PIN). As mentioned, unlike traditional exact-match caches in which a cache hit saves a disk/network IO, cache hits in GraphCache may result into vastly different reductions in query processing times. One of the reasons why this is so, is that cache hits reduce the candidate set of the coming query by possibly vastly different amounts. However, neither *LRU* nor *POP* (actually, none of the known replacement policies) take this into account. This gives rise to the next, exclusive to GraphCache, replacement policy: PIN (short for Popularity and sub-Iso test Number) Instead of looking just at the number of hits H of a cached query, PIN assigns each cached graph a utility value equal to R/A, where R is the total number of subgraph isomorphism tests alleviated by said cached query, and A is the same aging factor as above. The utility formula of PIN can also be rewritten as: $\frac{R}{A} = \frac{H}{A} \cdot \frac{R}{H}$, which can be interpreted as the probability of the query being a hit (i.e., its popularity), times the average savings in number of subgraph isomorphism test per hit. In our running example, this policy would evict queries 13 and 91.

PIN + Sub-Iso Tests Costs (PINC). PIN takes into account the number of sub-iso tests alleviated. Another GraphCache-exclusive replacement policy PINC further considers the possibly vast differences in query execution times. PINC assigns each cached query a utility value equal to C/A, where A is the same aging factor as above, and C is the total decrease in query processing time due to the cached query. Alas, this figure cannot be computed unless the relevant sub-iso tests are actually performed, which is a moot point in our case; instead, as mentioned (§5.2), we use a heuristic to estimate this cost. PINC may improve upon PIN's utility value computation by considering the actual (estimated) time cost of alleviated sub-iso tests instead of deeming them all equivalent. PINC's utility formula can be rewritten as: $\frac{C}{A} = \frac{H}{A} \cdot \frac{R}{H} \cdot \frac{C}{R}$, interpreted as the probability of a cached graph being hit, times the average savings in number of subiso tests per hit, times the average estimated time cost per saved sub-iso test. In our running example, PINC would evict queries 53 and 82.

The Hybrid Dynamic Policy (HD). As the cost component in PINC is only an estimation, using it does not always lead to improvements in GC's net query processing time. As a matter of fact, we have observed through a large number of experiments, that when the values of the R utility component exhibit a high variability, they are discriminative enough on their own. In such cases, taking the estimated cost into account can actually lead to lower time gains (i.e., PIN performing better than PINC). However, when the values of R exhibit a low variability, adding in the C component leads to considerable query processing time improvements.

Thus, the last replacement policy considered in this work (also exclusive to GraphCache), coined the *hybrid* policy (HD), coalesces both PIN and PINC. More specifically, when the HD policy is invoked, it first retrieves the R component from GC$_{stats}$ and computes its variability[25] by using the (squared) coefficient of variation (CoV). CoV is defined as the ratio of the (square of the) standard deviation over the (square of the) mean of the distribution. When $CoV > 1$, the associated distribution is deemed of high variability, as exponential distributions have $CoV = 1$ and typically hyper-exponential distributions (which capture many high-variance, heavy tailed distributions) have $CoV > 1$. In this case, HD performs cache eviction using PIN's scoring scheme; otherwise, it uses PINC's scoring scheme.

In our running example, the mean R value is $\mu = 161$ and its standard deviation $\sigma \approx 126$; then $CoV = \sigma/\mu \approx 0.78 < 1$ and thus HD will use PINC and evict queries 53 and 82.

7. PERFORMANCE EVALUATION

7.1 System Setup

We have implemented all aforementioned components and subsystems of GraphCache in Java over ≈6,000 lines of code. Experiments were performed on a Dell R920 host (4 Intel Xeon E7-4870 CPUs (15 cores each), with 320GB of RAM and 4×1TB disks, running Ubuntu Linux 14.04.4LTS.

We used GraphCache on top of three subgraph FTV and three SI methods (due to space limitations, we only present results for subgraph queries). The default value for the upper limit on the sizes C of the Cache and W of the Window stores were $C = 100$ and $W = 20$ respectively; we also experimented with other values for both C (200, 300) and W (50, 100, 200) to test their impact on GC's performance. Last, the sizes of the various thread pools are all set to 1 so as to show just the benefits of using a graph query cache. For the FTV methods we chose GraphGrepSX [2] (GGSX), Grapes [6], and CT-Index [14], specifically because they are proven to be top performers in their class[12]. Grapes and GGSX were configured to index paths up to length 4, and CT-Index to index trees up to size 6 and cycles up to size 8 using 4,096-bit-wide bitmaps. For Grapes, we examine two alternatives, Grapes1 and Grapes6, with 1 and 6 threads respectively. To be fair, we altered the code of Grapes so to stop query processing after the first match in each dataset graph. Please note that all mentioned values match their default configurations in [2, 6, 14]. For the SI methods we used GraphQL[10] as provided by [18] and a modified version of VF2[4] (denoted VF2+) provided by [14], again for being well-established and good performers[9, 12]; we also used vanilla VF2[4] since it has been used by several FTV implementations [2, 6, 9]. GC uses the Java Native Interface to directly execute the native C++ implementations of Grapes, GGSX, GraphQL and VF2, while CT-Index and VF2+ are implemented in Java and thus invoked directly from GC. This diversity in the implementation languages of the incorporated methods attests to GC's flexibility.

7.2 Datasets and Query Workloads

We employ three real-world (AIDS, PDBS, PCM) and one synthetic graph datasets with different characteristics. More specifically, AIDS[24] – the Antiviral Screen Dataset of the National Cancer Institute – contains topological structures of 40,000 molecules. Graphs in AIDS contain on average ≈45 vertices (std.dev.: 22, max: 245) and ≈47 edges (std.dev.: 23, max: 250) each, whereby the few largest graphs have an order of magnitude more vertices and edges. PDBS[11] is a dataset of graphs representing DNA, RNA and proteins, consisting of fewer (600) but larger graphs compared to AIDS, with on average ≈2,939 vertices (std.dev.: 3,215, max: 16,341) and ≈3,064 edges (std.dev.: 3,261, max: 16,781) per graph. PCM[32] consists of 200 graphs representing protein interaction maps, with on average ≈377 nodes (std.dev.: 187, max: 883) and ≈4,340 edges (std.dev.: 1,912, max: 9,416) per graph. Last, the Synthetic dataset was created using [3] and contained 1,000 graphs with on average ≈892 nodes (std.dev.: 417, max: 7,135) and ≈7,991 edges (std.dev.: 5.09, max: 8,007) per graph. We created this dataset as a larger counterpart to the PCM dataset, consisting of 5× more graphs, each being 2-3× larger on average than the average PCM graph. Graphs in AIDS and PDBS have low average node degree (AIDS ≈2.09, PDBS ≈2.13), whereas graphs of PCM and Synthetic have much higher average node degrees (PCM ≈22.39, Synthetic ≈19.52).

We follow the established principle for the generation of our workloads, using two different algorithms to synthesize queries from the dataset graphs, outlined below.

Type A Workloads. Queries in these workloads are generated in the following manner: first, a source graph is selected randomly from the dataset graphs; then, a node is selected randomly in said graph; finally, a query size is selected uniformly at randomly from several pre-defined sizes and a BFS is performed starting from the selected node. For each new node, all its edges connecting it to already visited nodes are added to the generated query, until the desired query size is reached. For the first two random selections above, we have used two different distributions; namely, Uniform (U) and Zipf (Z), with the probability density function of the latter given by $p(x) = x^{-\alpha}/\zeta(\alpha)$, where ζ is the Riemann Zeta function[26]. Ultimately, we had three categories of Type A workloads: "UU", "ZU" and "ZZ", where the first letter in each pair denotes the distribution used for selecting the starting graph, and the second for the starting node.

Type B Workloads (with no-answer queries). These workloads are generated as follows. For each of the query sizes, we first create two query pools: a 10,000-query pool with queries with non-empty answer sets against the dataset, and a second 3,000-query pool with no match in any dataset graph (i.e., empty result set). Queries for the first pool are extracted from dataset graphs by uniformly selecting a start node across all nodes in all dataset graphs, and then performing a random walk till the required query graph size is reached. Generation of no-answer queries has one extra step: we continuously relabel the nodes in the query with randomly selected labels from the dataset, until the resulting query has a non-empty candidate set but an empty answer set against the dataset graphs. Once the query pools are filled up, we generate workloads by first flipping a biased coin to choose between the two pools (with the "no-answer" pool selected with probability 0%, 20% or 50%), then randomly (Zipf) selecting a query from the chosen pool. We thus have three categories of Type B workloads: "0%", "20%" and "50%", denoting the above probability used.

We use Zipf $\alpha = 1.4$ by default; we also use $\alpha = 1.1$ representing a smaller skewness and $\alpha = 1.7$ for a higher skewness. As a reference point, web page popularities follow a Zipf distribution with $\alpha = 2.4$ [26]. Query graphs are generated in different sizes: 4, 8, 12, 16 and 20-edge graphs for the smaller AIDS and PDBS datasets; 20, 25, 30, 35 and 40-edge queries for the larger PCM and Synthetic datasets (as almost half of the dataset graphs in AIDS contain no more than 40 edges, larger queries are not usable). Such sizes are typical in the literature [6, 14, 35]. Workloads for AIDS and PDBS consist of 10,000 queries, while workloads for PCM and Synthetic contain 5,000 queries for practical reasons, as PCM/Synthetic queries take much longer to execute. We only allow one for one Window (i.e., 20 queries) before starting measuring GC's performance.

We report on both the benefits and the overheads of GC. Reported metrics include query time and number of sub-iso tests per query, along with the speedups introduced by GC. Speedup is defined as the ratio of the average performance (query time or number of sub-iso tests) of the base Method M over the average performance of GC when deployed over Method M (i.e., speedups >1 indicate improvements). The results were produced over more than 6 million queries! As a yardstick, [21] (also a cache but for XML databases) report a query time speedup of 2.6× with 10,000-query workloads generated using Zipf $\alpha = 1.5$, and a 1,500-query warm-up.

7.3 Results and Takeaways

Figure 4 depicts the speedups attained by GraphCache when CT-Index and GGSX where used as Method M (results for other FTV and SI methods showed similar trends and are thus omitted for space reasons). We can see that GraphCache attains significant speedups (up to 10× lower query processing times in this case), and that it is always one of the GC-exclusive policies (PIN, PINC) that produces the best results. A more subtle observation, though, is that there are cases where PIN wins over PINC and vice-versa; for example, PIN dominates the scene for queries against the AIDS dataset but it is PINC that takes the lead when querying the PDBS dataset. Ultimately, different cache replacement policies exhibit different performance depending on the workload and dataset characteristics. The question then is how to choose a replacement policy when said characteristics are unknown a-priori. Our answer to this question then, and the first takeaway message, is: ***When in doubt, use the HD replacement policy***, as it always manages to do better or on par with the best of the alternatives. For the remainder of this section we will be using HD as the replacement policy; results for other caching policies show similar trends and are thus omitted for space reasons.

Figure 5 depicts speedups in query processing time against all FTV methods for queries on the PDBS dataset (results for other datasets are similar). Query processing time speedups range from 1.60× (i.e., 37.5% lower processing time) to more than 42×. A similar picture is drawn in Figure 6 for speedups in the number of sub-iso tests performed. Juxtaposing Figure 5 and 6 leads to the following interesting insight: ***Reductions in the number of sub-iso tests do not translate directly into reductions in query time***; this validates our claim that cache hits in GraphCache render different benefits. In all cases, though, ***GraphCache achieves significant improvements in both query processing time and number of sub-iso tests performed***.

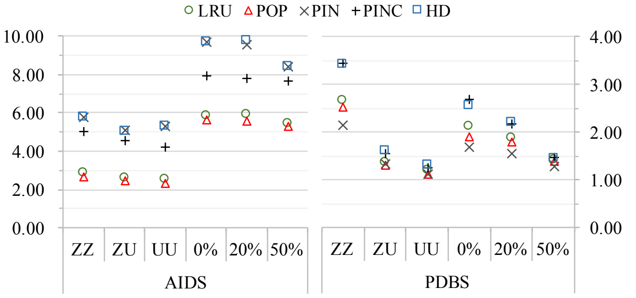
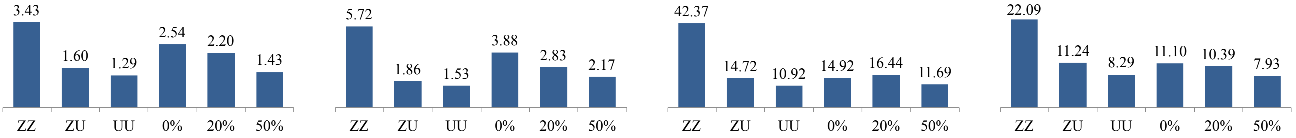


Figure 4: Query Processing Time Speedups over CT-Index Across Replacement Policies

Figure 7 shows the speedups achieved by GraphCache for Type B workloads against the AIDS dataset, for various values of the Zipf α skewness parameter (results for number of sub-iso tests and other workloads show similar trends and are omitted for space reasons). We can see that ***the more skewed the query distribution, the higher the gains from caching***. This is, of course, expected and has been shown times and again in related work on traditional caches, as caches are built on the premise of (temporal) locality of reference and thus more skewed query distributions have the potential to translate to higher hit ratios. A subtler, but equally important observation here, reached by examining Figure 5 in the light of the above result, is that ***GraphCache leads to significant performance gains even for query workloads with uniform query popularity distributions***. These distributions represent worst-case scenarios for caching schemes, but we can see speedups from 1.29× (≈20% lower times) up to ≈11× for the UU workloads, emphasizing a significant characteristic of GraphCache where the realm of "locality" is extended by subgraph/supergraph matches among queries, in addition to the traditional exact-match of isomorphic queries.

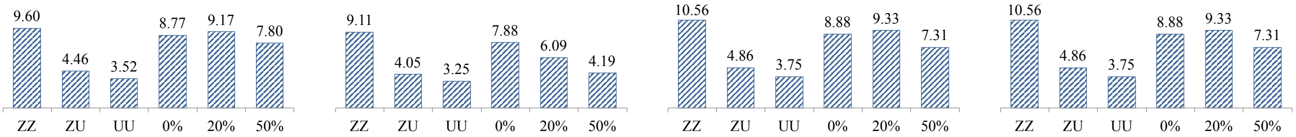
Figure 8 shows the performance of GC against GGSX for queries on AIDS and PDBS, for varying cache sizes (results for other methods and datasets show similar trends). We can see that ***increasing the cache size improves the performance of the cache***. However, this does not mean that one can increase the size of the cache indefinitely; the size of the cache is first limited by the amount of main memory available for GC, then by the overhead associated with updating the cache contents (more on this shortly).

Figure 9 shows the speedups in query time (9(a)) and number of sub-iso tests (9(b)) against Grapes6 for the PCM and Synthetic datasets, attained when the cache admission control is disabled (C) and enabled ($C + AC$). For clarity, performance without specific notes refer to turning off the cache admission control (C) by default. We can see that ***cache admission control leads to even higher speedups***, thus validating our observation regarding cache pollution and the appropriateness of our "expensiveness"-based mechanism. A subtler observation is that the corresponding speedup in the number of sub-iso tests is reduced when cache admission control is enabled, as shown in Figure 9(b). For better understanding of this trend, let us concentrate on



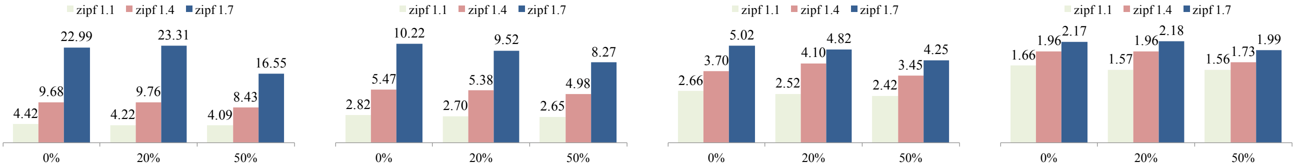
(a) CT-Index (b) GGSX (c) Grapes1 (d) Grapes6

Figure 5: GraphCache Speedup in Query Time for PDBS Across All Methods M



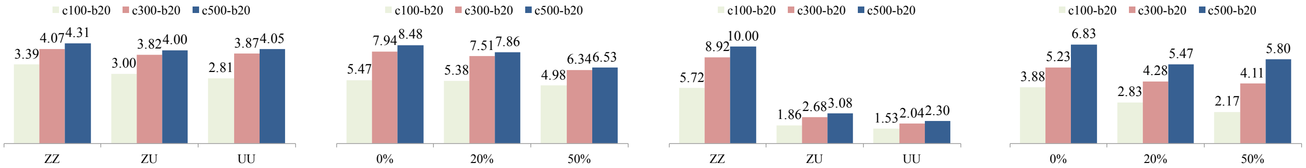
(a) CT-Index (b) GGSX (c) Grapes1 (d) Grapes6

Figure 6: GraphCache Speedup in Number of Sub-iso Tests for PDBS across all Methods M



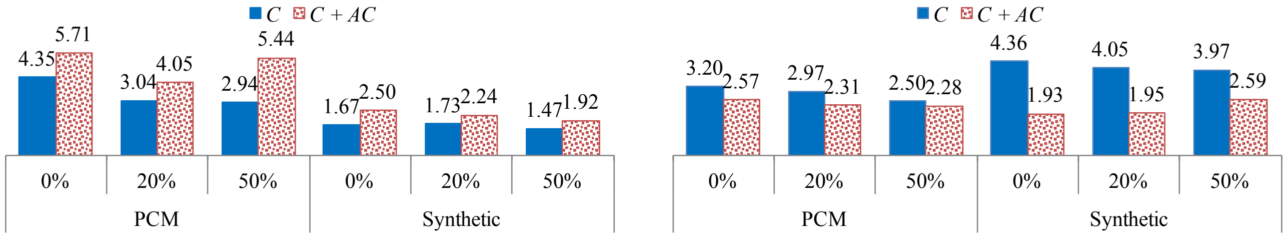
(a) CT-Index (b) GGSX (c) Grapes1 (d) Grapes6

Figure 7: GraphCache Speedup in Query Time for Type B workloads on the AIDS dataset, for various value of Zipf α



(a) AIDS/Type A Workloads (b) AIDS/Type B Workloads (c) PDBS/Type A Workloads (d) PDBS/Type B Workloads

Figure 8: GraphCache Speedup in Query Time against GGSX with various cache sizes



(a) Query Time Speedups (b) Speedups in Number of Sub-Iso Tests

Figure 9: GraphCache Performance vs Grapes6 for Type B Workloads on PCM/Synthetic Datasets

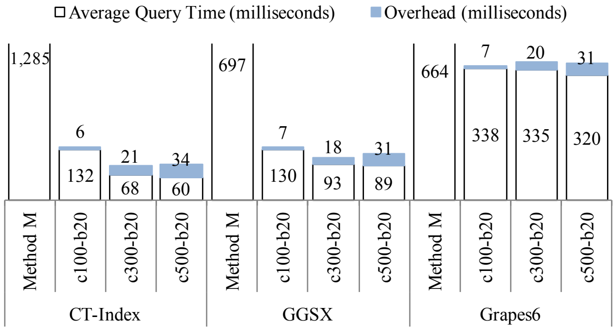


Figure 10: Average Execution Time and Overhead (milliseconds) per Query for the 20% workload on AIDS

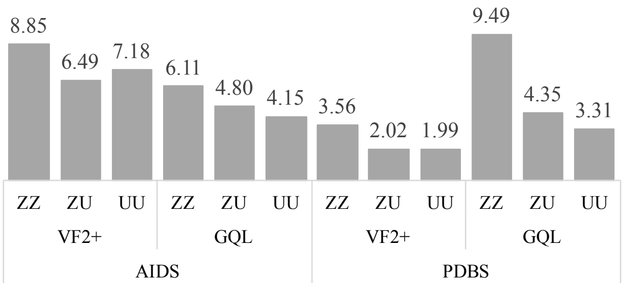


Figure 11: GC Query Time Speedups vs SI Methods

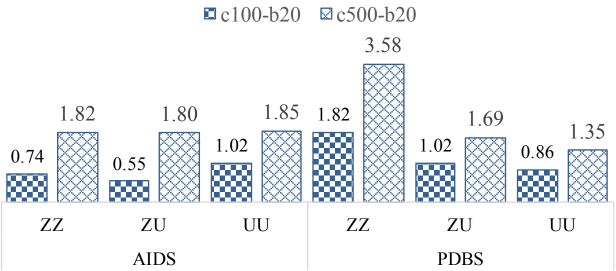


Figure 12: Query Time Speedups of GC/VF2+ vs CTIndex

the Synthetic-50% workload: GC without admission control yields a speedup as high as ≈4× in the number of sub-iso tests, but the resulting query time speedup is only ≈1.5×. The reason is that top expensive queries do not benefit as much when the cache is *polluted*: more specifically, the average time for the top-1% most time-consuming queries is ≈16.5 seconds with Grapes6, going down to ≈15 seconds for GraphCache without admission control – a 1.1× speedup; the remaining 99% "inexpensive" queries enjoy speedups of 2×, going from ≈0.200 seconds down to ≈0.100 seconds, but they account for a much smaller percentage of the overall query processing time compared to the top-1% ones. When we enable the admission control mechanism, these top-1% expensive queries are prioritized, with their average query processing time going down considerably to ≈10 seconds – a much improved 1.65× speedup. Hence, ***despite the lower speedup in number of sub-iso tests, the overall query processing time benefits greatly***.

We have shown so far that GraphCache leads to significant decreases in the query processing time and number of sub-iso tests of FTV (and SI) methods. We know that sub-iso tests take up the majority of the query processing time for FTV methods. A logical consideration, then, would be to try and increase the filtering power of these methods so as to further decrease the size of the resulting candidate set. This can be accomplished by increasing the size of the features recorded by FTV methods; larger features bear higher discriminative power as, obviously, the larger a feature the less its occurrences in dataset graphs. To this end, we reconfigured all FTV methods increasing their feature sizes by just one (i.e., max path length of 5 for Grapes and GGSX; trees of size 7, cycles of size 9, and 8192 bits per bitmap for CT-Index). This minimal increase in feature size indeed led to better performance, with the average query processing time going down by approximately 10%; however, it also led to an almost doubling of the space required for the FTV indexes across all methods. At the same time, GC accomplishes its speedup for a ***negligible space overhead***; for example, for the AIDS dataset the memory and disk space required by GraphCache was just over 1% of the space required for the indexes of the various FTV methods, but leading to time speedups of up to 40× (figure omitted for space reasons).

Figure 10 depicts a break-down of query processing time for FTV methods and GraphCache, showing how much of GC time is spent (on average) to update the Window and Cache data stores (including executing the cache replacement algorithms and re-indexing the cached query graphs), for various cache sizes. As we can see, ***the time overhead for cache maintenance chores is trivial***. Another interesting observation is that, although increasing the size of the cache improves query processing time (as also shown in Figure 8), it also leads to an increase in the overhead associated with the maintenance of the cache contents. For the cache sizes considered in this work we can see that what we lose in maintenance overhead, we gain in query time. That means that, if we had designed our architecture to update the cache contents in-line (i.e., not in parallel) with query processing, we would see diminishing returns with larger cache sizes. Our current design does not suffer from this problem; however, we expect that, for considerably larger cache sizes, this overhead may outgrow the time required for the Window to fill (and thus for a new replacement/re-indexing round to begin). The upside is, though, that ***even with the meagre cache sizes used in this work, the performance gains are enough to not warrant a much larger cache***.

Figure 11 depicts the query processing speedups of GC over the two well-established SI methods considered in this work – GQL and VF2+ (vanilla VF2 results where similar and are omitted for space and readability reasons). We can see that ***GC improves the performance of well-established SI methods***, with the same meagre 100-query cache configuration as above. ***This is significant in that GC provides a new way to expedite sub-iso tests (as opposed to developing yet another SI heuristic) which is usable with any mainstream SI method.*** Note the interesting finding that VF2+ speedup for AIDS UU workload is close to that of AIDS ZU (7.18 vs 6.49), whereas one might have expected a different outcome. Intuitively, the ZU workload bears more exact-match hits than UU, due to the skewness of selecting source graphs during query generation (see §7.2). And it does: we measured circa 2.5X the number of exact-match cache hits in ZU vs UU. However, recall that GC exploits also sub/supergraph hits. When exact-matches are not frequent, GC loads graphs in the cache that can help with their sub/supergraph relationships. Indeed, we measured circa 2X such matches for the UU workload vs ZU. Of course, the overall performance result is a very complex picture and depends on how big benefit is each saved exact-match vs each saved sub/supergraph match. But the key insight here is that ***by utilizing exact-matches and sub/supergraph matches, GC can introduce significant benefits in both skewed and non-skewed workloads***.

Let us now take a step back and look at how FTV methods and GC operate: they both expedite queries by filtering out dataset graphs, thus producing a reduced candidate set. The logical question then is: what happens if we pitch a full-blown FTV method against GC operating on top of a simple SI method? Figure 12 shows the results when comparing GC on top of VF2+ against CT-Index (also using VF2+ for its verification chores), across several datasets and Type-A workloads (results for Type-B workloads omitted for space reasons). For the small 100-query cache, GC performs on par or better than CT-Index in six out of nine cases, slightly worse in two other cases, and takes up to double the time of CT-Index in the remaining worst case. Note, though, that GC's space requirements are under ≈15% of the space requirements of CT-Index's index for PDBS and under 0.2% for AIDS, and that CT-Index has the fastest verification algorithm and by far the smallest index among all FTV methods considered in this work. The situation is more impressive when using the larger (500-query) cache, where GC matches or outperforms CT-Index across the board (by a factor of 1.8× on average). Note that even for this "larger" cache, GC's space requirements are less than

≈70% of CT-Index's index size for PDBS and less than 1% for AIDS (and comparable to the latter against GGSX and Grapes). The conclusion is then that ***GC can replace the best-performing FTV methods, achieving comparable or better performance for a fraction of the space and no pre-processing cost*** as no indexing is needed.

8. CONCLUSIONS

We presented GraphCache, to the best of our knowledge the first full-fledged caching system for general subgraph/supergraph query processing, including its architecture meeting demanding design goals, a number of GC-exclusive graph-query-aware cache replacement policies, and an accompanying cache admission control mechanism. The proposed system can be used to expedite all current FTV and SI methods (bridging these two, alas, separate threads of research so far), and is applicable for both subgraph and supergraph queries. Our extensive performance evaluation has proven the applicability and appropriateness of our approach. GC achieves considerable improvements in query processing time for meagre space overheads. Our work also revealed a number of key lessons, pertaining to graph caching and query processing. Future work currently focuses on two big ticket items: first, to develop a distributed/decentralized version of GraphCache; second, to extend GraphCache to benefit subgraph queries when finding all occurrences of a query graph against a single massive stored graph.

9. REFERENCES

[1] A. Balmin et al. A framework for using materialized XPath views in XML query processing. In *Proc. VLDB*, 2004.

[2] V. Bonnici et al. Enhancing graph database indexing by suffix tree structure. In *Proc. IAPR PRIB*, 2010.

[3] J. Cheng, Y. Ke, and W. Ng. GraphGen. http://www.cse.ust.hk/graphgen/.

[4] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE TPAMI*, 26(10):1367–1372, 2004.

[5] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[6] R. Giugno et al. Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. *PloS One*, 8(10):e76911, 2013.

[7] A. Gubichev and T. Neumann. Exploiting the query structure for efficient join ordering in SPARQL queries. In *Proc. EDBT*, 2014.

[8] W.-S. Han, J. Lee, and J.-H. Lee. Turbo$_{ISO}$: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proc. SIGMOD*, 2013.

[9] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. iGraph: A framework for comparisons of disk-based graph indexing techniques. *PVLDB*, 3(1-2):449–459, 2010.

[10] H. He and A. K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proc. SIGMOD*, 2008.

[11] Y. He et al. Structure of decay-accelerating factor bound to echovirus 7: a virus-receptor complex. *PNAS*, 99:10325–10329, 2002.

[12] F. Katsarou, N. Ntarmos, and P. Triantafillou. Performance and scalability of indexed subgraph query processing methods. *PVLDB*, 8(12):1566–1577, 2015.

[13] J. Kim, H. Shin, and W.-S. Han. Taming subgraph isomorphism for RDF query processing. *PVLDB*, 8(11):1238–1249, 2015.

[14] K. Klein, N. Kriege, and P. Mutzel. CT-index: Fingerprint-based graph indexing combining cycles and trees. In *Proc. ICDE*, 2011.

[15] G. Koloniari and E. Pitoura. Partial view selection for evolving social graphs. In *Proc. GRADES*, 2013.

[16] L. Lai et al. Scalable subgraph enumeration in MapReduce. *PVLDB*, 8(10):974–985, 2015.

[17] L. V. S. Lakshmanan et al. Answering tree pattern queries using views. In *Proc. VLDB*, 2006.

[18] J. Lee et al. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, 2012.

[19] K. Lillis and E. Pitoura. Cooperative XPath caching. In *Proc. SIGMOD*, 2008.

[20] B. Lyu et al. Scalable supergraph search in large graph databases. In *Proc. ICDE*, 2016.

[21] B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *Proc. VLDB*, 2005.

[22] M. Martin, J. Unbehauen, and S. Auer. Improving the performance of semantic web applications with SPARQL query caching. In *Proc. ESWC*, 2010.

[23] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *Proc. SIGMOD*, 2008.

[24] NCI - DTP AIDS antiviral screen dataset. http://dtp.nci.nih.gov/docs/aids/aids_data.html.

[25] R. Nelson. *Probability, Stochastic Processes, and Queueing Theory*. Springer Verlag, 1995.

[26] M. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46:323–351, 2005.

[27] N. Papailiou, D. Tsoumakos, P. Karras, and N. Koziris. Graph-aware , workload-adaptive SPARQL query caching. In *Proc. SIGMOD*, 2015.

[28] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *PVLDB*, 8(5):617–628, 2015.

[29] K. Semertzidis and E. Pitoura. Durable graph pattern queries on historical graphs. In *Proc. ICDE*, 2016.

[30] Z. Sun et al. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, 2012.

[31] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[32] C. Vehlow et al. CMView: Interactive contact map visualization and analysis. *Bioinformatics*, 27:1573–1577, 2011.

[33] J. Wang, J. Li, and J. X. Yu. Answering tree pattern queries using views: a revisit. In *Proc. EDBT*, 2011.

[34] J. Wang, N. Ntarmos, and P. Triantafillou. Indexing query graphs to speedup graph query processing. In *Proc. EDBT*, 2016.

[35] X. Yan et al. Graph indexing: a frequent structure-based approach. In *Proc. SIGMOD*, 2004.

[36] D. Yuan, P. Mitra, and C. L. Giles. Mining and indexing graphs for supergraph search. *PVLDB*, 6(10):829–840, 2013.

Subgraph Querying with Parallel Use of Query Rewritings and Alternative Algorithms

Foteini Katsarou
School of Computing Science
University of Glasgow, UK
f.katsarou.1@
research.gla.ac.uk

Nikos Ntarmos
School of Computing Science
University of Glasgow, UK
nikos.ntarmos@
glasgow.ac.uk

Peter Triantafillou
School of Computing Science
University of Glasgow, UK
peter.triantafillou@
glasgow.ac.uk

ABSTRACT

Subgraph queries are central to graph analytics and graph DBs. We analyze this problem and present key novel discoveries and observations on the nature of the problem which hold across query sizes, datasets, and top-performing algorithms. Firstly, we show that algorithms (for both the decision and matching versions of the problem) suffer from straggler queries, which dominate query workload times. As related research caps query times not reporting results for queries exceeding the cap, this can lead to erroneous conclusions of the methods' relative performance. Secondly, we study and show the dramatic effect that isomorphic graph queries can have on query times. Thirdly, we show that for each query, isomorphic queries based on proposed query rewritings can introduce large performance benefits. Fourthly, that straggler queries are largely algorithm-specific: many challenging queries to one algorithm can be executed efficiently by another. Finally, the above discoveries naturally lead to the derivation of a novel framework for subgraph query processing. The central idea is to employ parallelism in a novel way, whereby parallel matching/decision attempts are initiated, each using a query rewriting and/or an alternate algorithm. The framework is shown to be highly beneficial across algorithms and datasets.

CCS Concepts

•**Information systems** → **Database query processing;**
•**Mathematics of computing** → *Graph algorithms;*

Keywords

Graph databases, graph query processing, subgraph isomorphism

1. INTRODUCTION

Graphs are ideal for representing complex entities and their relationships/interactions and subgraph querying is essential to graph analytics. In *subgraph querying*, given a pattern graph (query) and a graph DB, we want to know whether it is contained in each DB graph (the decision problem) and/or find all its occurrences within it (the matching problem). Subgraph querying entails the subgraph isomorphism problem (abbreviated as sub-iso), which is NP-complete. Subgraph querying has received a lot of attention. Related work is categorized in two major categories: the *filter-then-verify (FTV)* and the *no-filter, verify (NFV)* methods. Numerous methods have been proposed for the problem and three recent experimental analysis papers ([7, 9, 12]) compare and stress-test proposed methods.

In this work, we conduct a comprehensive analysis of this problem. Our analysis aims to (i) lead to interesting novel findings about the nature of the problem and existing solutions, (ii) analyse and quantify said discoveries and their effect on well-established existing solutions, and (iii) show that the findings can be used to develop a framework that can offer large performance gains. Specifically, we first recognize the existence of "straggler" queries; i.e., queries whose execution time is dramatically higher than the rest. This holds for all query workloads and all datasets examined and across all tested FTV and NFV algorithms. Subsequently, we reveal and quantify the interesting fact that isomorphic instances of queries can have a wild variation in querying times. Then we generate isomorphic instances of the original query using statistics on vertex-label frequencies and/or vertex degrees and we investigate their performance. Moreover, for NFV methods in particular, we additionally show that challenging queries are algorithm-specific, with a straggler query for one algorithm possibly being easy for others. Finally, we incorporate these findings in a novel framework, coined the Ψ-framework, that exploits parallelism for both FTV and NFV methods, achieving large performance gains. Specifically, instead of trying to come up with new algorithms for sub-iso testing, we utilize isomorphic query rewritings and existing alternative algorithms in parallel. Extensive experimentation shows that our framework can be highly beneficial across datasets and workloads, and for both FTV and NFV methods.

2. BACKGROUND

2.1 Related work

Related work is categorized in two major categories. In the first category, proposed methods typically address a *decision* problem, where given a dataset of many (typically small) graphs and a query/pattern graph q, the method decides whether q is contained in any graph in the dataset.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.04

Most of the so-called *filter-then-verify (FTV)* or *indexed subgraph query processing* methods solve this decision problem, and work in 2 stages. In the index construction phase, stored graphs are decomposed into features which are then indexed, along with graph-id lists; i.e., lists of graphs that contain the feature. During query processing, query graphs are similarly decomposed into features; graphs from the dataset that do not contain one or more of these features definitely do not contain the query and are thus pruned away. The remaining graphs form the *candidate set.* At the verification stage, the query graph is tested for subgraph isomorphism against each graph in the candidate set to produce the final answer. The target of all these methods is to prune the candidate set and thus to reduce the number of sub-iso tests performed. Related works can be classified along 4 major dimensions: (i) type of indexed features (where "feature" refers to substructures of indexed graphs used to produce the index, independently of whether these are actually stored in the index or not): paths [1, 5, 30], trees [15, 25], simple cycles, or graphs [3, 20, 21, 22, 24, 29]; (ii) approach for extracting said features from indexed graphs: i.e., exhaustive enumeration [1, 10, 20, 30] or frequent subgraph mining techniques [3, 21, 22, 24, 25, 29]; (iii) index data structure: hash table, tree, trie; and (iv) whether the index stores location information or not. FTV methods are extensively discussed in [7, 9]. In [9] we concluded that Grapes[5] and GGSX[1] are the best solutions in terms of index construction and query processing time, and scalability limitations.

In the second category, proposed methods address a *matching* problem, whereby sub-iso testing is performed to find *all* the embeddings of the query graph q in a given large, stored graph g without performing any graph filtering in advance. We will call them the *no-filter, verify (NFV) methods.* Proposed methods, apart from the sub-iso test, additionally comprise of a pre-processing step where they maintain a feature-based index consisting of: (i) vertices and edges [15, 18], (ii) shortest paths [28] or (iii) subgraphs [8, 26] up to a certain size. The algorithms store vertex label lists along with additional information to facilitate the sub-iso test. A number of such methods were presented and compared in [12], concluding that (i) although there was no single algorithm to outperform all others in all occasions, GraphQL[8] was the only one that managed to complete all the tested query workloads; (ii) all three of GraphQL, sPath[28] and QuickSI[15] showed very good performance; but also that (iii) all existing algorithms have weaknesses in the way they apply their join selection and pruning heuristics, leading to the need for new graph matching algorithms.

There is nothing obstructing the NFV methods being applied for the decision problem and the FTV methods for the matching problem. FTV methods were originally proposed to work with datasets consisting of numerous, relatively small graphs, and their effectiveness relies on their achieved filtering, whereas NFV methods construct an index primarily to locate candidate vertices of the query in a large stored graph. For the current work, we opt to utilize all proposed methods for the originally proposed problems.

TwinTwig[11] and sTwig[16] deal with very large graphs, stored in a distributed infrastructure, and rely on parallel computing to perform sub-iso testing. Within FTV methods, iGQ[19] is a recent approach that employs caching on top of any proposed FTV method to improve performance. Semertzidis et al. [14] considered pattern queries over time-evolving graphs, which are beyond the scope of this study. Finally, there has been considerable work on the subject of *approximate graph pattern matching.* Related techniques (e.g. [10, 17, 20, 23, 27]) perform subgraph matching, but with the support for wildcards and/or approximate matches. All of these algorithms are not directly related to our work as we focus on exact subgraph matching.

As subgraph querying is an important problem, we expect that many researchers will keep focusing on trying to improve upon existing algorithms in the future. Indeed, since the publication just a few years ago of [12], comprehensively comparing the then state of art, newer algorithms have been proposed [6] with better performance. Nonetheless all algorithms show exponential execution times even at small query sizes (up to 10 edges)[13]. Our contributions aim to help this process in two ways. First, by revealing key insights, based on comprehensive experimentation, about the problem itself and how they affect well-known algorithms. Second, by shedding light onto a novel overall approach to the problem and its benefits. Namely, instead of focusing solely on developing new solutions by improving earlier algorithms, try to benefit from the wealth of ideas already existing within previous algorithms! Specifically, our findings show that different algorithms are appropriate for different queries. Furthermore, they show that different query rewritings are appropriate for different queries and for different algorithms! Finally, the existence of straggler queries poses new challenges for the performance comparison of different algorithms, needing more detailed performance metrics and experimenting with more challenging queries. All current works miss the above points: (i) they only consider one query rewriting, if at all, for all queries, (ii) they use only one algorithm for all workload queries, and (iii) they do not stress-test their algorithms with more challenging queries (e.g., larger sizes). Our framework shows that such misses also lead to misses of dramatic performance improvements.

2.2 Definitions

Definition 1 (Graph). *A graph $G = (V, E, L)$ is defined as the triplet consisting of the set $V = \{v_i\}, i = 1, ..., n$ of vertices of the graph, the set $E \subseteq \{(v, u) : v, u \in V\}$ of edges between vertices in the graph, and a function $L : V|E \rightarrow \mathcal{L}$ assigning a label $l \in \mathcal{L}$ ($\mathcal{L}$ being the set of all possible labels) to each vertex $v \in V$ and each edge $e \in E$.*

We assume that each node in a graph is assigned an integer in the interval $[1, n]$, so that no two nodes in a graph have the same number; we call this the node ID.

Definition 2 (Graph Isomorphism). *Two graphs $G = (V, E, L)$ and $G' = (V', E', L')$ are isomorphic iff there exists a bijection $I : V \rightarrow V'$ that maps each vertex of G to a vertex of G', such that if $(u, v) \in E$ then $(I(u), I(v)) \in E'$, $L(u) = L'(I(u))$, $L(v) = L'(I(v))$, and vice versa.*

Note that, given a graph G, a graph G' isomorphic to G can be trivially produced by permuting the node IDs in G.

Definition 3 (Subgraph Isomorphism). *A graph $G = (V, E, L)$ is subgraph isomorphic to a graph $G' = (V', E', L')$, denoted by $G \subseteq G'$, iff there exists an injective function $I : V \rightarrow V'$ such that if $(u, v) \in E$ then $(I(u), I(v)) \in E'$ and $L(u) = L'(I(u))$ and $L(v) = L'(I(v))$. Graph G is then called a subgraph of G'.*

Much like all of the works mentioned earlier, we focus on non-induced subgraph isomorphism.

3. EXPERIMENTAL SETUP

3.1 Short description of used Algorithms

3.1.1 FTV methods

Both Grapes[5] and GGSX[1] index the simplest form of features – i.e., paths – up to a maximum length. Paths are searched in a DFS manner and indexed in a trie or suffix tree respectively. Compared to GGSX, Grapes takes an additional step and maintains location information. Also, Grapes features multi-threaded design for both indexing and query processing. In query processing, maximal paths of the query are extracted to form the query index which is matched with the dataset index, pruning away unmatched branches. Subsequently, the search space is further pruned by the frequencies of indexed features. After this step, GGSX forms its candidate set of graphs that will undergo sub-iso testing. Grapes further exploits the maintained location information to extract relevant connected components of the dataset graphs, against which sub-iso testing is performed.

The underlying isomorphism algorithm for both Grapes and GGSX is VF2[4]. VF2 does not define any order in which query vertices are selected. Given a query graph q and a dataset graph g, the algorithm chooses a vertex from q to match to vertices in g, and proceeds by then trying to match still unmatched vertices adjacent to the matched ones in q. Given an unmatched vertex in q, the set of candidate vertices of g is defined as the set of all vertices in g with the same label as the unmatched vertex in q. VF2 then employs 3 pruning rules to reduce the number of candidate vertices. The first rule removes candidates that are not directly connected to the already matched vertices of g. The second rule removes all candidates for which the number of adjacent unmatched nodes which are also adjacent to matched nodes of g, is smaller than the corresponding figure for the matched vertex of q. The final rule removes all g candidates with less adjacent (matched/candidate) nodes than the corresponding figure in q.

3.1.2 NFV methods

In the sub-iso test of QuickSI[15] (QSI for short), priority is given to the vertices with infrequent labels and infrequent adjacent edge labels. In the indexing phase, QuickSI precomputes the frequencies of labels and edges and uses them to compute the "average inner support" of a vertex or an edge; i.e., the average number of possible mappings of the vertex or edge in the graph. The inner support is later used in the graph matching process to assign weights on the edges of the query graph and construct a rooted minimum spanning tree (MST). In case of symmetries, edges are added in such a way that will make the MST denser. The order in which vertices are inserted to the MST defines the order in which they are then matched in the sub-iso test.

In the indexing phase of GraphQL[8] (GQL for short), the labels of all vertices along with the neighbourhood signatures, which capture the labels of neighbouring nodes in a radius i in lexicographical order, are indexed. In the subgraph matching phase, the algorithm starts by retrieving all possible matches for each node in the pattern. Subsequently, 3 rules are applied in order to prune the search space. First, the indexed vertex labels and neighbourhood signatures are used to infeasible matches. Then a pseudo subgraph isomorphism algorithm is applied to the problem iteratively up to level l; i.e., for every pair of possible graph-query vertex matches, the nodes adjacent to the query node should be matched to the corresponding neighbours of the graph. Finally, the algorithm needs to optimize the search order in the query before proceeding with the actual sub-iso test, which in turn consists of a number of *joins* of the candidate node lists. This optimization is based on an estimation of the result-set size of intermediate joins, and as it would be very expensive to enumerate all possible search orders, only left-deep query plans are considered.

sPath[28] (SPA for short), similarly to GraphQL, also maintains a neighbourhood signature comprised of shortest paths organized in a compact indexing structure. Specifically, in order to reduce the storing space, shortest paths are not really maintained, but they are decomposed in a distance-wise structure. In the query processing, the query is initially decomposed in shortest paths that are then matched to the candidate shortest paths from the stored graph. From all possible candidate shortest paths, those that (i) can cover the query and (ii) provide good selectivity, i.e. minimize the estimated result-set size of each join operation, are selected as candidates. For each one of the selected paths, an edge-by-edge verification is then used to perform the sub-iso test.

3.2 Setup

Experiments with Grapes and GGSX were conducted on a small cluster consisting of 5 nodes, each featuring an Intel Core i5-3570 CPU (3.4GHz, 4 physical cores, 6MB cache), 16GB of RAM, 500GB disk per node, and running Ubuntu Linux 14.04. Experiments with QuickSI, GraphQL and sPath (i.e., the NFV methods) were conducted on a Windows 7 SP1 host, with 2 Intel Xeon E5-2660 CPUs (2.20GHz, 20MB cache) with 8 cores/16 vcores per CPU, 128GB of RAM, and 3.5TB disk. For practical purposes, we allowed a maximum limit of 10 mins for each query to be processed. Beyond that time, the execution is terminated and we proceed with the next query in the workload. Please note that this 10' limit does not apply in the indexing phases of the algorithms.

For Grapes and GGSX we used the implementations provided by their respective authors. However, in the case of Grapes, we had to alter the source code so that the VF2 verification step returns after the first match of the query graph, as opposed to the original implementation which was returning all possible matches. The reason for this is that FTV methods are mainly designed to retrieve the graphs that contain the query as an answer. For QuickSI, GraphQL and sPath, we used the implementation provided by [12].

We used the default values for the input parameters of the compared algorithms, as they were defined by their respective authors in the relevant publications and/or in their implementation code. More specifically:

- For GGSX and Grapes, we enumerated paths of up to size of 4.
- We ran Grapes with 1 and 4 threads; results for executions with 1 (resp. 4) threads are denoted by Grapes/1 (resp. Grapes/4).
- For GraphQL, we used a refined level of iterations of pseudo-subgraph isomorphism $r = 4$.
- For sPath, we used a neighbourhood radius of 4 and maximum path length 4.

		PPI	Synthetic
Dataset	# graphs	20	1000
	#disconnected graphs	20	0
	#labels	46	20
Per Graph	Avg #nodes	4942	1100
	StdDev #nodes	2648	483
	Avg #edges	26667	12487
	Avg density	0.0022	0.020
	Avg degree	10.87	24.5
	Avg #labels	28.5	20

Table 1: Dataset characteristics for FTV methods

	yeast	human	wordnet
#nodes	3112	4674	82670
#edges	12519	86282	120399
Avg degree	8.04	36.91	2.912
StdDev degree #nodes	14.50	54.16	7.74
Density	0.00258	0.0079	0.000035
#labels	184	90	5
Avg frequency labels	127	240	16534
StdDev frequency labels	322.5	430	152

Table 2: Dataset characteristics for NFV methods

- For QuickSI, GraphQL and sPath the number of searched embeddings of the pattern graph on the stored graph is capped at 1000; i.e., after finding the first 1000 matches, the algorithms terminate.

3.3 Datasets

We have chosen datasets which (a) have also been used by other studies, so as to enable possible direct comparisons, and (b) have key characteristics covering a large part of the design space (e.g., regarding graph size and density).

Table 1 summarizes the characteristics of the datasets that we used for the FTV methods. PPI (used in [5, 9]) is a real dataset representing 20 different protein-protein interaction networks. The majority of existing real datasets that were used for the FTV methods comprise of relatively small and sparse graphs. In [9] we showed that, for such datasets, both Grapes and GGSX perform adequately well. For our current study we are further interested in more challenging datasets and we thus employ an additional synthetic dataset generated with GraphGen[2], allowing various parameters of interest to be specified; namely, number of graphs, average number of nodes and density per graph, number of labels in the dataset). A more detailed description of how GraphGen constructs the dataset can be found in [9].

Datasets used for the NFV methods consist of only one graph as the primary task of these methods is to find all occurrences of the pattern graph in the large stored graph. Table 2 summarizes the characteristics of the three real datasets – namely yeast, human and wordnet — that we have used for the NFV methods. Yeast and human were previously used in [12], while Wordnet[1] was used in [16].

3.4 Query Workloads

To generate each of the queries, first we select a graph from the dataset uniformly and at random, and from that graph we select a node uniformly and at random. Starting from said node, we generate a query graph by incrementally adding edges chosen uniformly at random from the set of all edges adjacent to the resulting query graph, until it reaches the desired size. For the synthetic dataset, we used 100 queries of size 24, 32 and 40 edges for Grapes/1 and Grapes/4. We did not run GGSX against the synthetic dataset, because of excessive amount of time required for the experiments to complete. For the PPI dataset, we used 100 queries of size 16, 20, 24, and 32 edges. For the NFV methods, we used 200 queries of 10, 16, 20, 24 and 32 edges. Last, for QuickSI we only report results against the yeast dataset, as (i) it was the easiest NFV dataset to process, and (ii) QuickSI always had many more cases, compared to GraphQL and sPath, where query processing exceeded the 10’ cap. For all used methods, the majority of the queries completed in under 2”. We call them *easy* queries. Another portion of queries had processing times in the 2” to 600” range; we denote these *2”-600”* queries. We use the term *completed* to refer to all queries that finished within the 10’ limit; those that did not are called *hard* or *killed.*

3.5 Performance Metrics

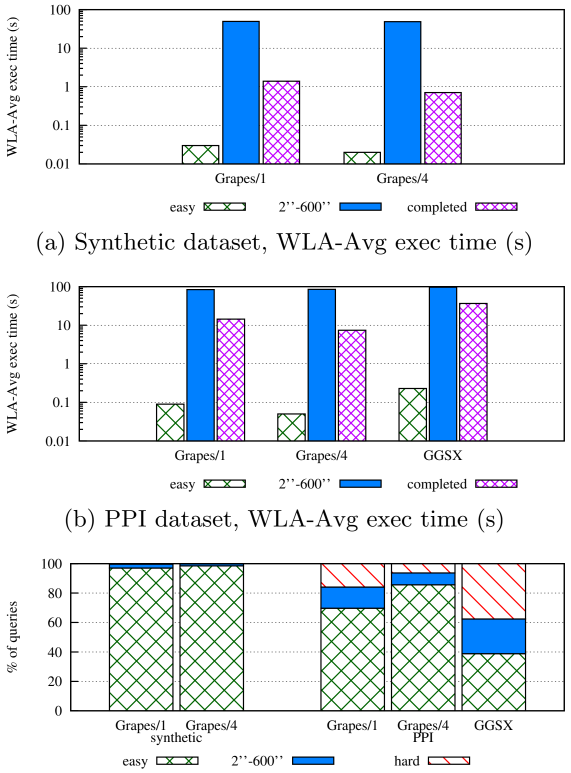
For every query against a stored graph, we measure the *Execution Time*, denoted *exec time*, for both FTV and NFV methods, while *avg exec time* denotes the average execution time. Specifically for FTV methods, this is the pure sub-iso time; i.e., excluding the index loading and filtering times, which add only a trivial overhead. For FTV methods reported times are in seconds, while for NFV methods times are in milliseconds, unless stated otherwise.

Let q_i be a given query and t_i^M the exec time of q_i over method M. Let also $q_{i,j}$ be the j-th isomorphic instance of q_i and $t_{i,j}^M$ the exec time of $q_{i,j}$ over method M. Finally, let $t_{i,j}^{\Psi}$ be the exec time of $q_{i,j}$ over our proposed Ψ-framework. We define the (max/min) metric as: $\frac{\max_j(t_{i,j}^M)}{\min_j(t_{i,j}^M)}$. The minimum value of this metric is 1, indicating that there are no variations between the min and max exec time. The higher the value of this metric, the higher the differences between the min and max exec time achieved by the isomorphic query instances. We also define the *speedup** metric as: $\frac{t_i^M}{T}$, where T is set to: (i) $\min_j(t_{i,j}^M)$, when comparing against the various isomorphic instances of q_i, (ii) $\min_M(t_{i,j}^M)$, when comparing against different methods, and (iii) t_i^{Ψ}, when comparing against our Ψ-framework. *speedup** represents what we lose in performance if we choose the original method over the various alternatives; i.e., *speedup** equals the maximum attainable speedup over the original method, if we chose the best of the examined alternatives. For comparison purposes, for queries that were killed at the 10’ limit we use this time (i.e., 600”) as their minimum execution time.

When comparing two sets of measurements $A = \{A_i\}$ and $B = \{B_i\}$, we can compute their average ratio in two ways:

- *Workload-Level Aggregation (**WLA**)*, given by $\frac{avg_i(B_i)}{avg_i(A_i)}$. When A and B contain query response times, the WLA computation would give the improvement in the overall average execution time. This metric is important from the *system* perspective as it encapsulates the overall performance change.
- *Query-Level Average (**QLA**)*, computed as $avg_i\left(\frac{B_i}{A_i}\right)$. When applied to query processing times, the QLA

[1] http://vlado.fmf.uni-lj.si/pub/networks/data/dic/Wordnet/Wordnet.htm



(a) Synthetic dataset, WLA-Avg exec time (s)

(b) PPI dataset, WLA-Avg exec time (s)

(c) Percentages of *easy*, *2"-600"*, and *hard* queries

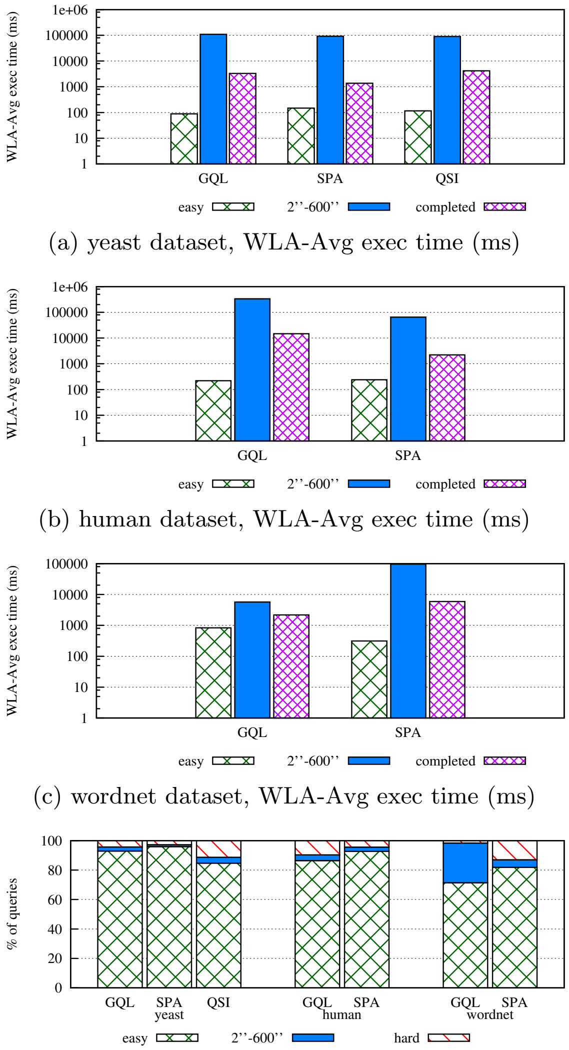
Figure 1: Stragglers in FTV methods

computation would give the average of per-query improvements. This metric is *user-centric* in the sense that each user cares what the performance improvement for his query is using different methods.

In both cases, $avg_i(X_i)$ is the average over all items X_i in the set X. Based on this distinction, the aforementioned (max/min) and *speedup** metrics can have a QLA or WLA version, denoted with a matching subscript; e.g., $speedup^*{}_{QLA}$. These two variants also carry over to other computations; for example, the standard deviation of the ratio of A and B would be computed as $\frac{stdDev_i(B_i)}{stdDev_i(A_i)}$ under WLA, and as $stdDev_i(B_i/A_i)$ under QLA. However, unless stated otherwise, we shall use QLA and WLA to denote averages.

4. STRAGGLER QUERIES

We know that as the dataset grows in terms of the size of graphs, query processing becomes harder; ditto as the size of the query graph increases [9]. But do these statements hold across all queries-dataset graph combinations? Running many queries against the whole dataset can hide the details of how much time is required per individual query-graph pair. In the case that a small portion of such pairs dominates the whole execution time, then by just looking at the whole query workload execution times it is easy to draw wrong conclusions about the algorithms' performance. Also, several related works choose to ignore queries whose execution is much higher compared to the rest. To investigate the above, in this study we execute each individual query against a single stored graph at a time.



(a) yeast dataset, WLA-Avg exec time (ms)

(b) human dataset, WLA-Avg exec time (ms)

(c) wordnet dataset, WLA-Avg exec time (ms)

(d) Percentages of *easy*, *2"-600"*, and *hard* queries

Figure 2: Stragglers in NFV methods

Observation 1: In all of workloads generated by us or found in other papers, our experiments show "stragglers"; i.e., queries whose processing time is many orders of magnitude higher compared to the rest.

In order to back our observation, we present our results from the experiments on the aforementioned datasets against both FTV and NFV methods (fig. 1 and 2).

4.1 FTV methods

Fig. 1 presents the results from the query workloads on the FTV methods. Specifically, 1(a) and 1(b) show the average execution times for the corresponding algorithms for the synthetic and the PPI dataset respectively (GGSX/synthetic results omitted; see §3.4). 1(c) presents the percentage of the sub-iso tests that were *easy*, *2"-600"*, and *hard* for both the synthetic and PPI datasets. As expected, Grapes/4 has a much smaller percentage of *killed* queries compared to Grapes/1 and GGSX. A notable thing here is that although for both Grapes/1 and Grapes/4 the percentage of *2"-600"*

		GraphQL	sPath	QuickSI
10-edge q	AET *easy* (ms)	66.84	134.78	131.67
	% of *easy*	100	99.5	99
	AET *2”-600”* (ms)	-	2871.44	50367.40
	% of *2”-600”*	0	0.5	1
	% of *hard*	0	0	0
32-edge q	AET *easy* (ms)	130.66	120.71	96.62
	% of *easy*	80	91	67.5
	AET *2”-600”* (ms)	140812	140781	78917.2
	% of *2”-600”*	6.5	3	6
	% of *hard*	13.5	6	26.5

Table 3: Results for NFV methods on the yeast dataset (AET: avg exec time)

		GraphQL	sPath
10-edge q	AET *easy* (ms)	179.49	209.91
	% of *easy*	100	98
	AET *2”-600”* (ms)	-	182392
	% of *2”-600”*	0	1
	% of *hard*	0	0
32-edge q	AET *easy* (ms)	246.31	277.13
	% of *easy*	71.5	84.5
	AET *2”-600”* (ms)	93523.7	31817
	% of *2”-600”*	4.5	4.5
	% of *hard*	24	11

Table 4: Results for NFV methods on the human dataset (AET: avg exec time)

queries is $< 5\%$ in the synthetic dataset and $< 10\%$ in PPI, the avg exec time across all completed queries is significantly affected; that is, the most expensive queries dominate the execution time.

4.2 NFV methods

Fig. 2 presents the results from the query workloads on the NFV methods (QuickSI human/wordnet results omitted; see §3.4), while tables 3 and 4 give results for 10- and 32-edge queries for the yeast and human datasets. We can use the 10-edge query results to compare our findings with those presented in [12]. [12] used small query sizes (up to 10 edges) and showed that the best performing algorithm is GraphQL, because it managed to complete all tested query workloads. With our experiments, we confirm this for both the yeast and human datasets and for queries of size 10 edges. GraphQL performs better compared to sPath, having also 0% of *hard* queries. The same holds for the *easy* queries of 32 edges. However, the picture is reversed when looking at the rest of the queries. In this case, the percentage of *killed* queries is double for GraphQL compared to sPath.

We note that unlike yeast and human where sPath performs overall better than GraphQL having (i) smaller avg exec times on the completed queries and (ii) smaller percentages of *hard* queries, in wordnet this behavior is reversed. Based on our analysis, it’s very difficult to claim that one algorithm is better than the other. In fact, in order to claim that, we need to define a performance metric of interest. Such a metric could be the percentage of *killed* queries, but note that it depends on the time limit imposed on query processing. For example, in wordnet, if the threshold was 2”, then sPath would be better than GraphQL, but if we change this threshold the picture changes.

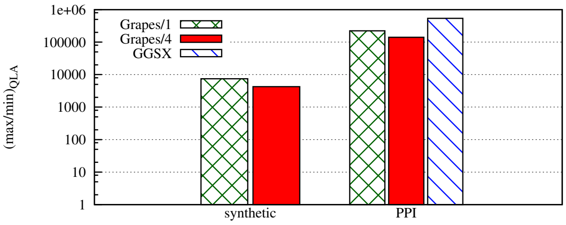


Figure 3: Avg $(max/min)_{QLA}$ for FTV methods

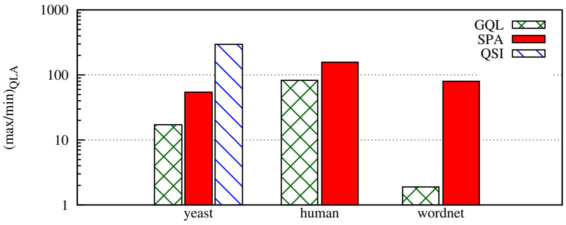


Figure 4: Avg $(max/min)_{QLA}$ for NFV methods

We summarize our results to the following 3 conclusions: (1) Some queries are *hard.* (2) Different algorithms have different percentages of *completed* queries; thus, different algorithms find different queries hard. (3) As the most expensive queries dominate the avg exec time, one must include a sufficient number of *hard* queries in order to draw conclusions about the relative performance of the algorithms.

5. ISOMORPHIC QUERIES

The proposed sub-iso methods ([8, 28, 15]), as well as [12] that compares them, claim that the search order on the query can have a huge impact on query processing time. We agree with this claim. In the current study, we take a further step and instead of relying on the order that the individual method imposes, we generated our own isomorphic query rewritings. To achieve this, we keep the structure of the query graph and the labels on the nodes unchanged, and permute the node IDs. Subsequently, we transform the query graph to an input format compatible with each individual method and perform the query processing. In the following experiments, we used a total of 6 different rewritings per query, leading to the following observation.

Observation 2: Queries which are isomorphic to the original query have widely and wildly different execution times.

We attribute this behavior to the fact that all proposed methods do not define a strict order in which the nodes of the query are matched, as computing a globally optimal join plan would be too computationally expensive. Thus, all methods rely on heuristics (see §3.1) in order to minimize the search space for the join plan.

5.1 FTV methods

Fig. 3 depicts the QLA average value of the (max/min) metric for the synthetic and PPI datasets, for the FTV methods (GGSX results omitted for the synthetic dataset; see §3.4). Table 5 additionally reports the stdDev, min, max and median values of $(max/min)_{QLA}$. In the calculations,

		Grapes/1	Grapes/4	GGSX
synthetic	stdDev	86,700.40	65,988.40	-
	min	1.06	1.02	-
	max	3,820,000.00	3,490,000.00	-
	median	3.90	4.45	-
PPI	stdDev	469,934	395,285	1,020,000
	min	1.03	1.02	1.01
	max	3,680,000	3,160,000	12,000,000
	median	1,186.51	11.19	109,086.00

Table 5: $(max/min)_{QLA}$ statistics for FTV methods

		GraphQL	sPath	QuickSI
yeast	stdDev	287.54	533.86	1685.71
	min	1.01	1.01	1.00
	max	7286.33	6695.85	15021.60
	median	1.40	1.36	1.61
human	stdDev	440.18	662.78	-
	min	1.00	1.04	-
	max	4115.06	4087.81	-
	median	1.82	1.96	-
wordnet	stdDev	20.55	396.87	-
	min	1.01	1.01	-
	max	646.44	3081.14	-
	median	1.21	1.34	-

Table 6: $(max/min)_{QLA}$ statistics for NFV methods

		Grapes/1	Grapes/4	GGSX
synthetic	stdDev	53,785.70	24,267.60	-
	min	1.00	1.00	-
	max	3,820,000	2,110,000	-
	median	1.36	1.24	-
PPI	stdDev	302,250	237,573	758,668
	min	1.00	1.00	1.00
	max	3,370,000	2,910,000	9,390,000
	median	3.71	1.67	1,751.22

Table 7: $speedup^*_{QLA}$ statistics for FTV methods across rewritings

		GraphQL	sPath	QuickSI
yeast	stdDev	235.61	422.56	1193.03
	min	1.00	1.00	1.00
	max	7286.33	6695.85	15021.60
	median	1.10	1.08	1.30
human	stdDev	259.93	492.45	-
	min	1.00	1.00	-
	max	4115.06	4087.81	-
	median	1.09	1.08	-
wordnet	stdDev	20.55	244.66	-
	min	1.00	1.00	-
	max	646.44	3081.14	-
	median	1.13	1.08	-

Table 8: $speedup^*_{QLA}$ statistics for NFV methods across rewritings

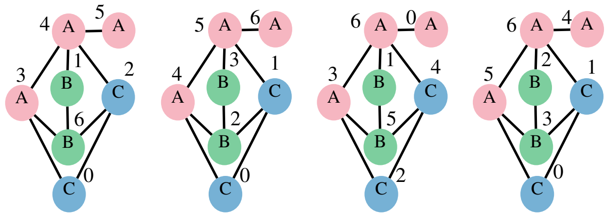
we did not include queries that were not helped by any of the isomorphic instances tried; i.e., queries that were *hard* on all tested isomorphic instances of the query. This behavior occurred in 0.0036% and 1.4% of queries for Grapes/1 on the synthetic and PPI datasets respectively, and 0.37% of queries for Grapes/4 and 1.96% of queries for GGSX for the PPI dataset. We note that the "max" and "average" values of $(max/min)_{QLA}$ are only lower-bound estimations, because of the 10' limit that we used instead of the actual verification time. In these results, we observe that there is an at least 6 orders of magnitude difference between the min and the max value of $(max/min)_{QLA}$, with the median (apart from GGSX) being closer to the min value. Along with the high stdDev, we can see that isomorphic instances of the same query can indeed have widely and wildly different verification times.

5.2 NFV methods

Fig. 4 reports the QLA-average values of the (max/min) metric for the yeast, human and wordnet datasets, for the tested NFV methods (QuickSI results omitted for the human and wordnet datasets; see §3.4). Table 6 reports the stdDev, min, max and median value of $(max/min)_{QLA}$. We report that 4.2%, 8.2% and 1.5% of queries were not helped by any tested isomorphic query instances for GraphQL and for yeast, human and wordnet respectively. For sPath the corresponding values are 2.1%, 1.4% and 11.8%. Finally, for QuickSI 8.6% of the queries were not helped for yeast.

The QLA-average (max/min) for the NFV methods is up to 3 orders of magnitude lower than that of the FTV methods. This is somewhat expected as the NFV methods define a more strict order in which the nodes of the query are matched and thus leave less space for wild variations. However, this order is still significantly affected by the initial node ids of the query, and thus we still see per-query (max/min) values of up to 2 orders of magnitude.

We summarize our overall results to the following conclusions: (1) For every isomorphic test to be executed, given a query graph q and a stored graph, there is an isomorphic version of q that can take anywhere from 2 to 6 orders of magnitude more time to execute compared to the least expensive version of the query. This holds across all algorithms and datasets tested. (2) Although the presented figures hide the details of the individual query sizes, we report that the harder the queries (higher query sizes), the higher these number are.

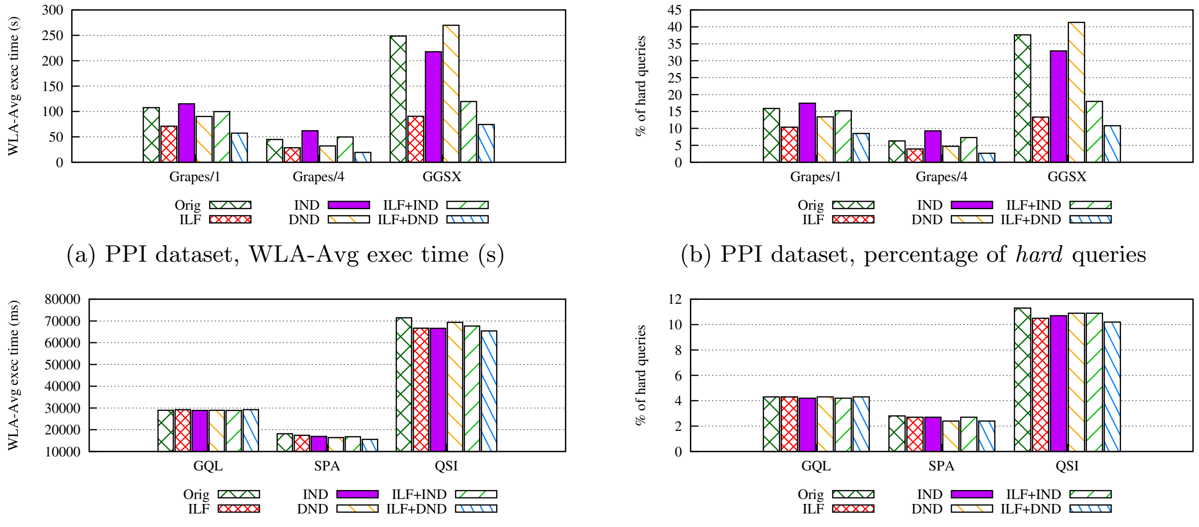


(a) Original (b) ILF (c) IND (d) ILF+IND

Figure 5: Isomorphic queries generated with different rewritings (assuming the label frequencies in the stored graph are: "A"=20, "B"=15, "C"=10)

6. GRAPH QUERY REWRITING

Having established that isomorphic versions of a query can have dramatically different execution times, we set out



(a) PPI dataset, WLA-Avg exec time (s)

(b) PPI dataset, percentage of *hard* queries

(c) yeast dataset, WLA-Avg exec time (ms)

(d) yeast dataset, percentage of *hard* queries

Figure 6: Results for individual query rewrtings for both FTV and NFV methods

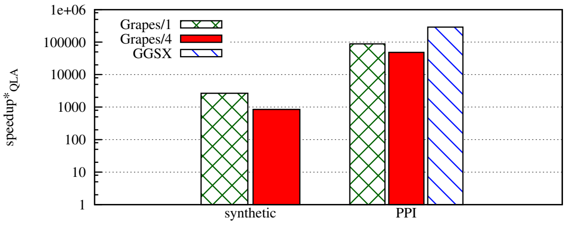


Figure 7: Avg $speedup^*_{QLA}$ for FTV methods across rewritings

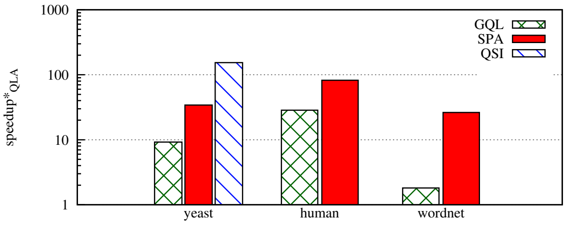


Figure 8: Avg $speedup^*_{QLA}$ for NFV methods across rewritings

to construct specific rewritings, constructing graphs isomorphic to the original queries, with the aim to capture these benefits. We have developed and experimented with several such query rewritings. We outline below five such rewritings, all performed by carefully permuting the node IDs in the query graph:

- `Query Rewriting ILF (`*`Increasing Label Frequency`*`)`: In a preprocessing step, we compute the frequencies of node labels in the stored graph, sorted in increasing frequency order. Given this order, we produce a rewriting of the query graph so if i, j are the node IDs of query graph nodes n_i, n_j, $L(n_i), L(n_j)$ are their labels, and $f(L(\cdot))$ is the frequency of a label $L(\cdot)$ in the stored graph, then $f(L(n_i)) < f(L(n_j)) \Rightarrow i < j$. Ties can appear in 2 cases: (i) two or more query nodes have the same label, or (ii) two or more query nodes have different labels but with the same frequency. These ties are broken arbitrarily.
- `Query Rewriting IND (`*`Increasing Node Degree`*`)`: The nodes of the query are sorted in increasing node degree order; i.e., if n_i, n_j are two query graph nodes, and $d(\cdot)$ is the degree (number of edges) of a node, then $d(n_i) < d(n_j) \Rightarrow i < j$. In the case of nodes with the same number of edges, ties are broken arbitrarily.
- `Query Rewriting DND (`*`Decreasing Node Degree`*`)`: This rewriting is similar to the IND but the nodes of the query are sorted in decreasing node degree and the nodes ids are assigned accordingly.
- `Query Rewriting ILF+IND`: This rewriting is the same as ILF above, with ties being broken in an IND manner: i.e., nodes with smaller outgoing degree get a lower node id.
- `Query Rewriting ILF+DND`: This rewriting is the same as ILF+IND, with ties being broken in a DND manner.

Fig. 5 presents an example of the above rewritings. Note that the ILF+IND rewriting in 5(d) is also a valid ILF rewriting. As we already mentioned, ties are (utterly) broken in an arbitrary way, and thus one may compute several different isomorphic graphs for the same rewriting.

Indicatively[2] and because of space restrictions, in fig. 6 we report the WLA average processing times of the original query and the 5 proposed query rewritings for the PPI and yeast datasets, as well as the corresponding percentages of the *hard* queries. For the FTV methods, the best performing rewritings are ILF and ILF+DND, with the percentage of *hard* queries being significantly improved. For the NFV methods, the picture is slightly different. GraphQL shows no considerable improvement with any individual rewriting; as a matter of fact, there are rewritings leading to higher

[2] We obtained similar results for the synthetic dataset for the FTV methods and the human dataset for the NFV methods. The sole exception was sPath, whose percentage of *hard* queries increased slightly for the wordnet dataset.

avg exec times than the original query. For sPath, the DND and ILF+DND rewritings reduced the percentage of *killed* queries from 2.8% to 2.4%. For QuickSI, ILF+DND reduced the percentage of *killed* queries from 11.3% to 10.2%, but DND only brought it down to 10.9%. More importantly, note that there is no single rewriting that manages to improve all algorithms across all datasets and workloads.

Observation 4: "Stragglers" can have isomorphic counterparts which are not stragglers.

Please note that the max and average reported $speedup^*$ represent a lower-bound estimation because of the value 600" that we use for the *hard* queries that were killed. Additionally, in our calculations we do not include the few queries that were killed for both the original instance and with all the rewritings of the query (see §5.1 and §5.2).

6.1 FTV methods

Fig. 7 presents the average $speedup^*{}_{QLA}$ for the FTV methods for the synthetic and PPI datasets (GGSX/synthetic results omitted; see §3.4). Additionally, table 7 reports the QLA stdDev, min, max and median of $speedup^*{}_{QLA}$. Moreover, as we increased the size of the queries, $speedup^*{}_{QLA}$ increased by up to 3 orders of magnitude (not visible in the figure as results are aggregated). For the presented results, median $speedup^*{}_{QLA}$ is close to min $speedup^*{}_{QLA}$, evidencing again a wide variation in the benefits of the isomorphic query rewritings. Keeping in mind that the majority of the queries are *easy* (fig. 1), we conclude that large performance gains can come from improving the *hard* queries.

6.2 NFV methods

Fig. 8 presents the average $speedup^*{}_{QLA}$ for the NFV methods for the yeast, human and wordnet datasets (QuickSI human/wordnet results omitted; see §3.4). Table 8 reports the stdDev, min, max and median of the $speedup^*{}_{QLA}$. The performance of sPath could seemingly be improved by one to two orders of magnitude across all datasets. The same holds for QuickSI on yeast. GraphQL could also be improved by more than a factor of 10× on the yeast and human datasets. However, no significant improvement was possible for GraphQL on wordnet. The reason why this is so, is somewhat subtle. Apart from what the algorithms are doing internally to match the query, other culprits are the characteristics of the actual stored graphs and the generated queries. Looking at the statistics of the graphs (table 2), yeast and especially wordnet are very sparse graphs with small average node degree. Thus, the majority of the generated queries are paths and the rewritings based on node degrees are not effective in this case. Additionally for wordnet, the small number of labels (only 5) and distribution of the frequencies of the labels being highly skewed leads to the generation of queries that in their majority contain only 1 or 2 labels, with the second label appearing only once. As a result, the rewritings are of little use in these cases.

7. ALGORITHM-SPECIFIC STRAGGLERS

As we already mentioned in section 4, we notice that for the NFV methods, different algorithms have different percentages of *hard* queries, leading to the conclusion that different algorithms find different queries hard. In this section we elaborate on this observation.

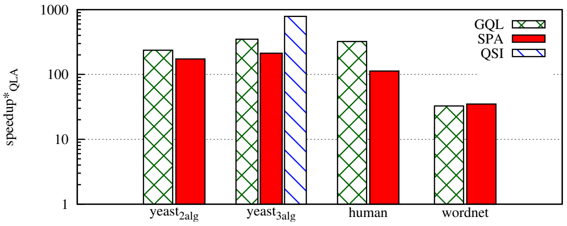


Figure 9: Avg $speedup^*{}_{QLA}$ when utilising different algorithms on NFV methods

		GraphQL	sPath	QuickSI
$yeast_{2alg}$	stdDev	1094.57	1051.65	-
	min	1.00	1.00	-
	max	9189.36	9129.60	-
	median	1.00	1.80	-
$yeast_{3alg}$	stdDev	1596.47	1255.34	2162.97
	min	1.00	1.00	1.00
	max	13060.10	12403.70	12312.70
	median	1.00	1.88	1.32
human	stdDev	1394.34	570.83	-
	min	1.00	1.00	-
	max	30873.80	4341.44	-
	median	1.00	1.04	-
wordnet	stdDev	253.56	104.42	-
	min	1.00	1.00	-
	max	3733.78	932.58	-
	median	2.47	1.00	-

Table 9: $speedup^*{}_{QLA}$ statistics when utilizing different algorithms on NFV methods

Observation 5: "Stragglers" are algorithm-specific; i.e., by evaluating the same query workloads with various algorithms, we have seen that a "straggler"-query for one algorithm can be a typical query for the other algorithms.

Fig. 9 presents the average $speedup^*{}_{QLA}$ for the yeast, human and wordnet datasets and for the tested algorithms. In table 9, we additionally report the stdDev, min, max and median of $speedup^*{}_{QLA}$. For the yeast dataset, we present the results with utilizing all 3 algorithms (noted as $yeast_{3alg}$), as well as with the pair of algorithms (GraphQL and sPath) that we utilize for the remaining datasets (noted as $yeast_{2alg}$). For the yeast dataset, all tested queries were helped by the use of different algorithms. In the human and wordnet datasets, only 0.8% and 0.1% of the queries were not helped by this scheme. Note that the $speedup^*{}_{QLA}$ values for using multiple algorithms are higher compared to the $speedup^*{}_{QLA}$ values achievable with multiple query rewritings (see §6.2). This leads to the conclusion that the use of multiple algorithms could be way more beneficial compared to the rewritings, which are not always effective (§6.2).

8. THE Ψ-FRAMEWORK

In this section we present how we incorporate our findings in a novel framework that exploits parallelism. The proposed framework is called Ψ-framework (**P**arallel **S**ubgraph **I**somorphism framework). Unlike recent related work [11, 16], by having different threads/machines working on different versions of the problem our Ψ-framework exploits parallelism in a novel way. We utilize Grapes and GGSX (as well as GraphQL and sPath) as well-established FTV (resp. NFV) methods. Within our Ψ-framework we have incorporated the original implementations of Grapes and GGSX as provided by their authors, and of GraphQL and sPath as found in [12].

In the FTV methods we leave intact the index construction and the filtering stages during query processing. In the verification stage, for every graph in the candidate set, we instantiate a number of threads equal to the number of the isomorphic-query rewritings we utilize. These threads run in parallel with each being assigned one rewriting of the initial query, and the first thread to finish is the "winner"; i.e., the rest of the threads are killed and the algorithm proceeds with the verification of the next graph in the candidate set.

Ψ-framework for the NFV methods works similarly to the verification stage of the FTV methods. However, we mentioned in observation 5 that stragglers disappear when using an alternative matching algorithm. We incorporate this finding in our Ψ-framework by running simultaneously two threads: one for sPath and one for GraphQL with the original query. Again after the completion of the fastest thread, the rest of them are killed.

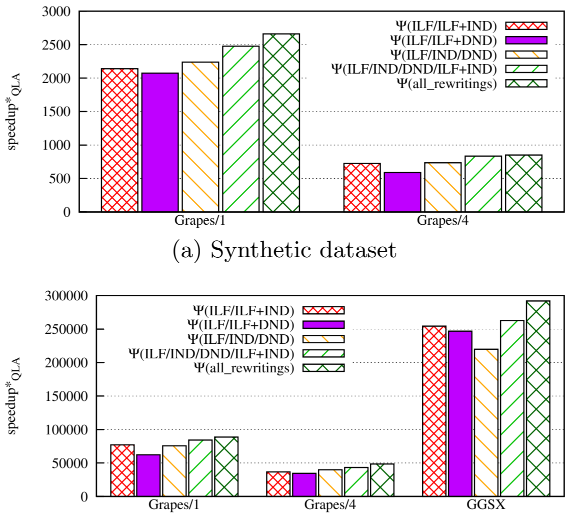
On one hand we have seen that the more the isomorphic instances we use, the better the speedup we gain in the graph matching process. On the other hand, the instantiation and synchronization of many threads come with a non-trivial overhead, impacting the overall speedup. To this end, in our performance evaluation we report on the speedup achieved by several beneficial combinations of rewritings. We note that our Ψ-framework is of course not the only solution to the straggler-queries' problem. Undoubtedly, it would be preferable to choose the right isomorphic query instance and/or algorithm to use to minimize the query execution time. However, given the complex nature of the sub-iso problem, we leave such design decisions for future work.

The cost of producing the query rewritings was measured from a few tens (for smaller query sizes) to a few hundreds (for the biggest query sizes) of μsecs; being a negligible overhead to the overall query processing time, we ignore it in the figures and omit any further discussion of this cost factor.

8.1 FTV methods

Fig. 10 and 11 present the avg $speedup^*_{QLA}$ and avg $speedup^*_{WLA}$ respectively for utilizing different versions of Ψ-framework on the FTV methods. Specifically, we present the avg $speedup^*_{QLA}$ and avg $speedup^*_{WLA}$ of the following versions of Ψ-framework: (a) ILF/ ILF+IND (2 threads), (b) ILF/ ILF+DND (2 threads), (c) ILF/ IND/ DND (3 threads), (d) ILF/ IND/ DND/ ILF+IND (4 threads) and (e) all 5 possible rewritings (5 threads). Our framework proves highly beneficial for all algorithms and datasets. Although not depicted in the figure, but as it was expected, by increasing the number of threads running multiple rewritings on the Ψ-framework, not only the avg execution time is significantly improved but also the percentage of hard queries is decreased, even leading to straggler-free executions. However, note that the Ψ-framework(ILF/ IND/ DND) (3 threads) is only 3-8% worse compared to Ψ-framework(ILF/ IND/ DND/ ILF+IND) (4 threads) for Grapes/1 and Grapes/4.

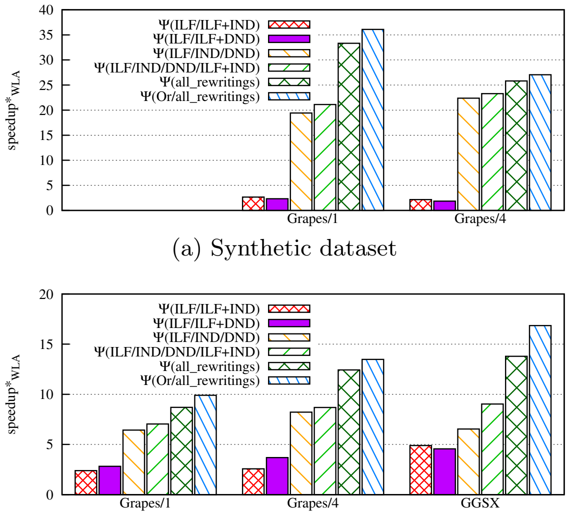
As Grapes is designed as a multi-threaded application, we additionally compare Grapes/4 against our Ψ-framework running Grapes/1 with the following four rewritings (for to-



(a) Synthetic dataset

(b) PPI dataset

Figure 10: Avg *speedup*$^*_{QLA}$ across different versions of our framework on the FTV methods



(a) Synthetic dataset

(b) PPI dataset

Figure 11: Avg *speedup*$^*_{WLA}$ across different versions of our framework on the FTV methods

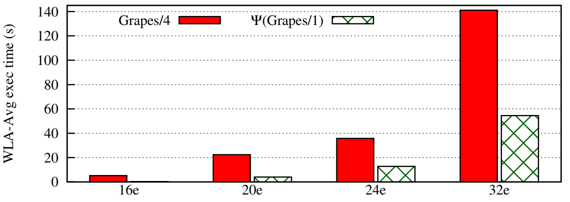
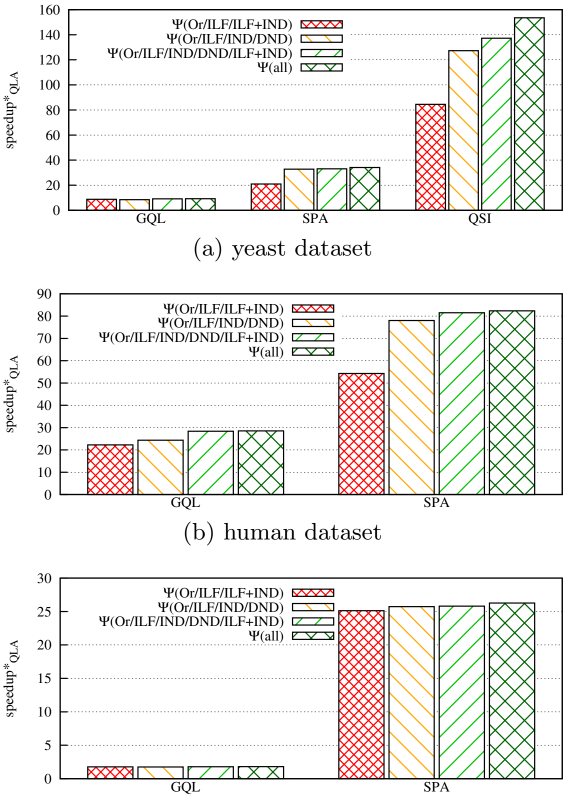


Figure 12: Comparison of avg exec time over the PPI dataset, for Grapes/4 against the Ψ-framework with 4 rewritings over Grapes/1



(a) yeast dataset

(b) human dataset

(c) wordnet dataset

Figure 13: Avg *speedup*$^*_{QLA}$ across different versions of Ψ-framework on the NFV methods

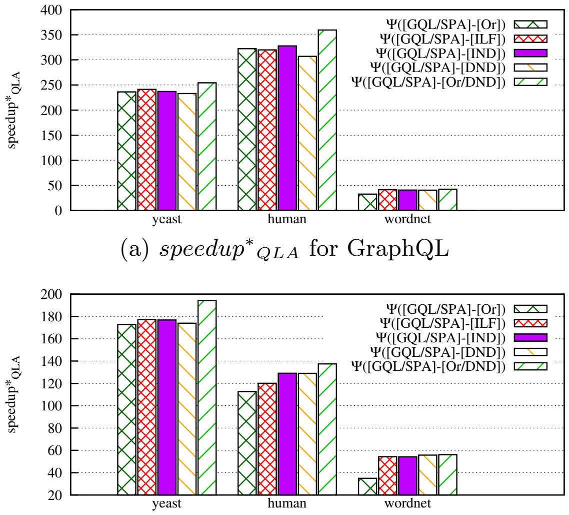
	PPI	yeast	human	wordnet
Grapes/4	6.29%	-	-	-
GraphQL	-	4.3%	10%	1.6%
sPath	-	2.8%	4.4%	13%
Ψ-fram	2.06%	0%	0.7%	0%

Table 10: Percentage of killed queries of FTV methods and our Ψ-framework

tal of 4 threads as well): ILF, IND, DND, ILF+IND. The results are presented in fig. 12 for the PPI dataset (results for the synthetic dataset were similar). Table 10 reports the percentage of killed queries for Grapes/4 and Ψ-framework on PPI. As is obvious, although both contenders have the same level of parallelism, Ψ-framework makes better use of its threads and leads to lower query processing times.

8.2 NFV methods

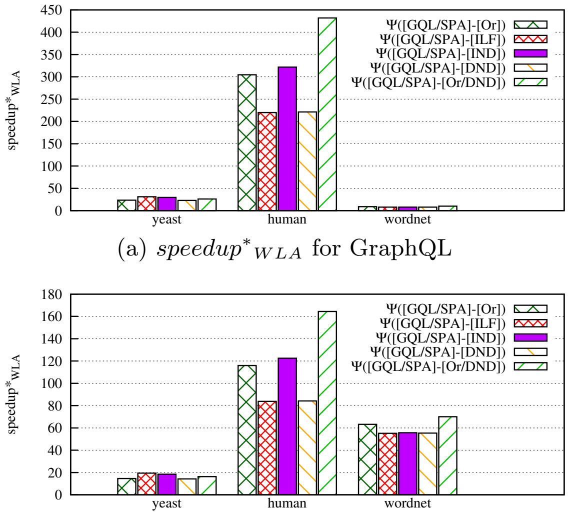
Fig. 13 presents the avg *speedup*$^*_{QLA}$ for utilizing different versions of Ψ-framework on the NFV methods (we omit the figures for avg *speedup*$^*_{WLA}$ due to space constraints). We utilize the following versions of Ψ-framework and the corresponding number of threads: (a) Orig/ ILF/ ILF+IND (3 threads) (b) Orig/ ILF/ IND/ DND (4 threads), (c) Orig/ ILF/ IND/ DND/ ILF+IND (5 threads), and (d) Orig + all-rewritings (titled as all) (6 threads). For all tested datasets and workloads, GraphQL benefited the least by the rewritings. The biggest improvements appear in the human dataset. We attribute this to the fact that this dataset comprises a denser graph with more labels, thus a larger portion of "hard" queries benefited by our rewritings and framework.



(a) *speedup*$^*_{QLA}$ for GraphQL

(b) *speedup*$^*_{QLA}$ for sPath

Figure 14: Avg *speedup*$^*_{QLA}$ for running multiple algorithms against NFV methods on Ψ-framework



(a) *speedup*$^*_{WLA}$ for GraphQL

(b) *speedup*$^*_{WLA}$ for sPath

Figure 15: Avg *speedup*$^*_{WLA}$ for running multiple algorithms against NFV methods on Ψ-framework

Finally, fig. 14 and 15 depict the avg *speedup*$^*_{QLA}$ and the avg *speedup*$^*_{WLA}$ for utilizing different algorithms and different versions of Ψ-framework on the NFV methods and on yeast, human and wordnet, against vanilla GraphQL and sPath respectively. We instantiated the following versions of our Ψ-framework with the corresponding number of threads: (a) GraphQL-Orig/ sPath-Orig (2 threads), (b) GraphQL-ILF/ sPath-ILF (2 threads), (c) GraphQL-IND/ sPath-IND (2 threads), (d) GraphQL-DND/ sPath-DND (2 threads). (e) GraphQL-Orig /sPath-Orig/ GraphQL-DND/ sPath-DND (4 threads). For both GraphQL and sPath, we were able to achieve up to 3 orders of magnitude improvement with our Ψ-framework on both per-query and per-workload metrics. Also, with the Ψ-framework, the percentage of hard queries was reduced and, for yeast and wordnet, hard queries became extinct – see Table 10.

9. CONCLUSIONS

We have studied the subgraph isomorphism problem, in both its decision and matching versions, using well-established FTV and NFV methods respectively, and against several different real and synthetic datasets of various characteristics. Our research has revealed and quantified a number of insights, concerning (i) the existence and role of straggler queries in a method's overall performance, (ii) the dramatically varying performance of isomorphic queries, (iii) the impressive impact that query rewriting can have when used before executing the query with several algorithms, and (iv) the fact that straggler queries are algorithm-specific. We suggested and used both WLA and QLA metrics to fully appreciate the performance of algorithms in the presence of stragglers. A number of query rewritings were proposed and our results showed that in many cases there existed one rewriting that could offer great performance advantages – with different rewritings being best for different queries. We showcased that, for the NFV algorithms, when a query was proved to be very expensive with one algorithm, another algorithm would actually manage to compute its answer very efficiently. These findings then naturally culminated into a novel framework, which employs in parallel different threads, each using a different well-known algorithm and/or a specific query rewriting, per query. This introduced dramatic improvements (up to several orders of magnitude) to FTV and NFV algorithms. We hope that our findings will open up new research directions, striving to find appropriate, per-query, isomorphic rewritings, in combination with alternate per-query sub-iso algorithms that can yield large improvements. Using machine learning models to predict which version of our framework (algorithms, rewritings) to employ per query is of high interest.

10. REFERENCES

[1] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha. Enhancing graph database indexing by suffix tree structure. In *Proc. IAPR PRIB*, 2010.

[2] J. Cheng, Y. Ke, and W. Ng. GraphGen. http://www.cse.ust.hk/graphgen/.

[3] J. Cheng, Y. Ke, W. Ng, and A. Lu. FG-index: towards verification-free query processing on graph databases. In *Proc. SIGMOD*, pages 857–872, 2007.

[4] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE TPAMI*, 26(10):1367–1372, 2004.

[5] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha. GRAPES: A software for parallel searching on biological graphs targeting multi-core architectures. *PloS One*, 8(10):e76911, 2013.

[6] W.-S. Han, J. Lee, and J.-H. Lee. Turbo$_{iso}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proc. SIGMOD*, 2013.

[7] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. iGraph: a framework for comparisons of disk-based graph indexing techniques. *PVLDB*, 3(1-2):449–459, 2010.

[8] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proc. SIGMOD*, pages 405–418, 2008.

[9] F. Katsarou, N. Ntarmos, and P. Triantafillou. Performance and scalability of indexed subgraph query processing methods. *PVLDB*, 8(12):1566–1577, 2015.

[10] K. Klein, N. Kriege, and P. Mutzel. CT-index: Fingerprint-based graph indexing combining cycles and trees. In *Proc. ICDE*, pages 1115–1126, 2011.

[11] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *PVLDB*, 8(10):974–985, 2015.

[12] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2), 2012.

[13] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *PVLDB*, 8(5):617–628, 2015.

[14] K. Semertzidis and E. Pitoura. Durable graph pattern queries on historical graphs. In *Proc. ICDE*, 2016.

[15] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.

[16] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, 2012.

[17] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *Proc. ICDE*, 2008.

[18] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the (JACM)*, 23(1), 1976.

[19] J. Wang, N. Ntarmos, and P. Triantafillou. Indexing query graphs to speedup graph query processing. In *Proc. ACM EDBT*, pages 41–52, 2016.

[20] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *Proc. ICDE*, pages 976–985, 2007.

[21] Y. Xie and P. Yu. CP-Index: on the efficient indexing of large graphs. In *Proc. CIKM*, 2011.

[22] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proc. SIGMOD*, pages 335–346, 2004.

[23] X. Yan, F. Zhu, P. S. Yu, and J. Han. Feature-based similarity search in graph structures. *ACM TODS*, 31(4):1418–1453, 2006.

[24] D. Yuan and P. Mitra. Lindex: a lattice-based index for graph databases. *VLDBJ*, 22(2):229–252, 2013.

[25] S. Zhang, M. Hu, and J. Yang. TreePi: A Novel Graph Indexing Method. In *Proc. ICDE*, 2007.

[26] S. Zhang, S. Li, and J. Yang. GADDI: Distance Index Based Subgraph Matching in Biological Networks. In *Proc. EDBT*, pages 192–203, 2009.

[27] S. Zhang, J. Yang, and W. Jin. SAPPER: subgraph indexing and approximate matching in large graphs. *PVLDB*, 3(1-2):1185–1194, 2010.

[28] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1-2):340–351, 2010.

[29] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta >= graph. In *PVLDB*, pages 938–949, 2007.

[30] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *Proc. EDBT*, pages 181–192, 2008.

Reverse k-Ranks Queries on Large Graphs

Yuqiu Qian, Hui Li, Nikos Mamoulis, Yu Liu, David W. Cheung
Department of Computer Science
The University of Hong Kong, Hong Kong
{yqqian,hli2,nikos,yliu4,dcheung}@cs.hku.hk

ABSTRACT

Given a collection of objects, the reverse k-ranks query takes as input a query object q in the set and returns the top-k objects that rank q higher compared to where other objects rank q. This query has been studied in the vector space, however, there is no previous work in the context of graphs. In this paper, we propose a filter-and-refinement framework, which prunes the search space while traversing the graph in search for the reverse k-ranks query results. We present an optimized algorithm and an index that apply on this framework and boost its performance. The proposed techniques are evaluated on real data; the experimental results show that our solutions scale well, rendering the query applicable for searching large graphs.

1. INTRODUCTION

Ranking queries (e.g., k-NN query [10], reverse k-NN query [13], reverse top-k query [21], reverse k-ranks query [27]) have become very popular in database management systems. Among them, the reverse k-ranks query has been recently proposed as an enhancement of the reverse top-k query, which ensures the same number of results for any query input. Specifically, given a *customer-product* vector space, where customers rank products, the reverse top-k query takes as input a product and an integer k, and produces as output the k customers that rank the product higher compared to its ranking by other customers. However, there is no prior work on how to evaluate reverse k-ranks queries on graphs, where graph proximity measures can be used to define the distance between nodes (and their ranking with respect to a query node).

Motivation. The reverse top-k query was extended to apply on large graphs in [26, 25]. Given a query node and an integer k, this query retrieves all other nodes that have the query node in the set of their k nearest nodes, based on a proximity measure. We conducted an experiment, where we apply reverse top-k queries (using shortest weighted path as the proximity measure) on the DBLP author collaboration graph [12]. Each node in the graph corresponds to an author and two authors are connected by an edge if they have published at least one paper together. Edges are weighted to reflect the strength of the collaboration [17, 11]. The application of a reverse top-k query is to let the query author know what other authors are keen to collaborate with him/her. According to our experiments in Section 6.2, for a large percentage of query nodes, the reverse top-k query returns either very few or too many results. The fact that reverse top-k queries do not have a fixed number of results limits their utility, especially in applications such as graph based recommender systems (e.g., tag recommendation [7], friendship recommendation [14], product recommendation [19] and paper recommendation [16]), where making recommendations to "cold-start" users who are weakly connected to the rest of the data is an important issue. On the other hand, the reverse k-ranks query returns a result of fixed size (k) for any query node; hence, it is particularly useful for new nodes of the graph (e.g., new social network users) who have little influence to other nodes and for "hot" nodes, which have very high influence but still want to shortlist the nodes that they are most attracted to them.

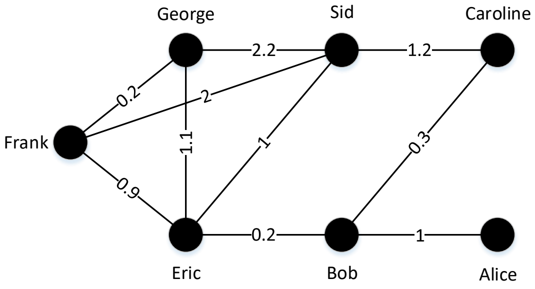


Figure 1: A Toy Example

EXAMPLE 1. *Figure 1 illustrates a toy example. Seven researchers form a weighted undirected graph. Alice is a new researcher and she only has a weak connection with Bob. If we use shortest path to measure proximity, we can get the rank matrix shown in Table 1. For example, Rank(Alice, Eric) is Eric's position in the list of nodes ordered by shortest path distance from Alice. Indeed, Eric is the 2nd closest node (after Bob) to Alice with a shortest path distance 1.2.*

A reverse top-k query having Alice as the query node with $k = 2$ *returns no results since Alice does not fall into any researchers' top-2 list (See first column of Table 1). This means that this query is not useful to Alice for recommending her researchers to collaborate with. On the other hand, a reverse* 2*-ranks query for Alice returns a nonempty result {Bob, Caroline}, since both Bob and Caroline rank Alice higher compared to her rank by other researchers, which means that these are the two researchers who are most likely to collaborate with her. If the query node is Eric and we use a re-*

Series ISSN: 2367-2005

10.5441/002/edbt.2017.05

verse top-2 query, we will recommend all other six researchers to him because of his close relationship to all of them; this result is overwhelming and would not be useful to Eric. On the other hand, a reverse 2-ranks query returns {Bob, Sid} (since Bob and Sid rank Eric as 1st while others rank him as 2nd).

Table 1: Rank Matrix

	Alice	Bob	Caroline	Sid	Eric	Frank	George
Alice	-	1	3	5	2	4	6
Bob	3	-	2	5	1	4	6
Caroline	4	1	-	3	2	5	6
Sid	6	2	2	-	1	4	5
Eric	6	1	2	4	-	3	5
Frank	6	3	4	5	2	-	1
George	6	3	4	5	2	1	-

Applications. In the era of big data, large volumes of graph data are becoming available. Reverse k-ranks queries over large-scale graphs can find application in spatial network data analysis, collaboration recommendation, dating, etc. For example, the management of a supermarket chain may want to investigate the space of potential customers. Given a road network which includes communities (i.e., estates) and supermarkets, a reverse k-ranks query will return a list of k communities which rank a given supermarket higher compared to where it is ranked by other communities, based on its network distance. The result can be used by the management to conduct targeted advertisement and promotion. As discussed, reverse k-ranks queries can also be used for collaboration and friendship recommendation in collaboration networks or social networks.

Contribution. In this paper, we study the evaluation of reverse k-ranks queries on large graphs, using shortest weighted path as the measure of proximity between nodes. To the best of our knowledge, there is no previous work on this problem. The special nature of graph data does not allow the application of the approaches proposed for reverse k-ranks queries in the vector space [27]. In addition, extending existing solutions for top-k search and reverse top-k search on graphs to compute reverse k-ranks queries is not trivial. Specifically, for a top-k search we only have to find the top-k proximity set of a single node q and for the reverse top-k search we need to compute the top-k sets of all nodes in the graph and check whether q appears in each of them. For the reverse k-ranks query, we must calculate all rank sets of all nodes in the graph and find the top-k ranks of q in them; therefore, a reverse k-ranks query is substantially more expensive than top-k search and reverse top-k search.

Our contributions can be summarized as follows:

- We study for the first time reverse k-ranks queries on large graphs and propose a filter-and-refine graph browsing framework to evaluate it. We propose effective bounds for the ranks of the examined nodes that limit the set of nodes which need to be accessed during query evaluation.
- We propose a dynamically refined, space-efficient index structure, which supports reverse k-ranks query evaluation. The index is paired with an efficient online query algorithm, which prunes a large number of nodes that are definitely in or not in the reverse k-ranks result and reduces the required refinements for the remaining candidates.
- We conduct an experimental study demonstrating the efficiency of our framework, as well as the effectiveness of the reverse k-ranks query in real graph applications.

The remainder of this paper is organized as follows. Section 2 provides a formal definition of reverse k-ranks search and discusses a baseline brute-force solution. In Section 3, we present a fundamental theorem and our basic two-step framework. Two efficient algorithms, Dynamic Bounded SDS-tree and Dynamic Bounded SDS-tree with index, are proposed in Section 4 and Section 5 respectively. Section 6 evaluates the effectiveness of reverse k-ranks queries and the efficiency of the proposed framework. In Section 7, we briefly discuss previous work related to reverse k-ranks queries. Finally, Section 8 concludes the paper.

2. PROBLEM DEFINITION

A formal definition of the reverse k-ranks query is given below:

DEFINITION 1. *(Rank(s,t)) Consider a weighted graph $G = (V, E)$, consisting of a set of nodes V and a set of edges E. Each edge in E carries a non-negative weight. For any two nodes $s, t \in V$, let $d(s, t)$ denote the shortest path distance from s to t, which is defined by summing up the weights of the edges along the shortest path from s to t. Let S be the set of nodes that satisfy $\forall p_i \in S, d(s, p_i) < d(s, t)$ and $\forall p_j \in (V - S - \{t\}), d(s, p_j) \geq d(s, t)$. Then, $Rank(s, t) = |S| + 1$ where $S \subset V$ and $|S|$ is the cardinality of S.*

DEFINITION 2. *(Reverse k-Ranks Query on a Graph) Given a weighted graph $G = (V, E)$, a query node q and a positive integer k, the reverse k-ranks query returns a subset T of V, such that $|T| = k$ and $\forall p_i \in T$, $\forall p_j \in (V - T - \{q\})$, $Rank(p_i, q) \leq Rank(p_j, q)$.*

Computing the reverse k-ranks set of a query node q is not trivial. A naive method, for each node $p_i \in V$, traverses the graph to find the distances to all other nodes from p_i in increasing order (i.e., using Dijkstra's algorithm) until q is encountered; this can give us Rank(p_i,q). During this process, the top-k of these ranks are maintained in a heap and eventually returned as results. Obviously, this method is very expensive. Another possible solution is to apply multiple reverse top-k' queries with an increasing k' value, until the number of results is similar to the k value of the reverse k-ranks query. This solution, apart from only giving an approximate result, is also expensive because the number of required reverse top-k' queries could be large and there is no straightforward method for evaluating them incrementally.

3. GENERAL TWO-STEP FRAMEWORK

To process reverse k-ranks queries efficiently, we design a two-step framework. First, we build a Shortest Distance Search tree based on the given query node and use it to prune the space of candidate nodes. Second, in a refinement step, we compute Rank(p_i,q) for each surviving candidate node p_i; during this process, the top-k nodes are maintained in a priority queue and they are finally output. Although our examples and illustrations are on undirected graphs, our solutions can directly be applied to directed graphs.

3.1 Filter step: SDS-Tree

Given a graph $G = (V, E)$, the Shortest Distance Search tree (SDS-tree) rooted at vertex q is a spanning tree T_q of graph G, such that the path distance from any other vertex p to q is the shortest path distance from p to q in G. SDS-Tree is similar to the Dijkstra tree [4], but on the transpose graph G^T, which can be different to G if G is directed. G^T is a directed graph on the same set of vertices as G, but with all of the edges of G reversed. That is, if G contains

Algorithm 1 Basic SDS-tree Construction

```
1: procedure ReverseKRank($q, G$)
2:     Priority Queue $Q \leftarrow \{q : 0\}$                ▷ nodes to visit
3:     $R \leftarrow \varnothing$                              ▷ reverse $k$-ranks result
4:     $D \leftarrow \varnothing$                              ▷ nodes visited
5:     $kRank \leftarrow Inf$                                 ▷ $k$-th top rank in $R$
6:     while $Q$ do
7:         $top \leftarrow Q.pop()$
8:         $D \leftarrow D \bigcup \{top\}$
9:         $top.rank \leftarrow GetRank(top, kRank)$
10:        if $top.rank \neq -1$ then                         ▷ Theorem 1
11:            Update $R$
12:            $kRank \leftarrow$ new $k$-th rank in $R$
13:            for t in top.neighbors() do
14:                if $t \notin D$ then
15:                    $dis \leftarrow top.dis + d[top][t]$
16:                    if $t \in Q$ and $t.dis > dis$ then
17:                        $t.dis \leftarrow dis$
18:                    else
19:                        $t.dis \leftarrow dis$
20:                        $Q.push(t)$
21:    return $R$
```

an edge (u, v) then G^T contains an edge (v, u) with same weight and vice versa. If G is undirected, then $G^T = G$.

To build the SDS-tree for the query input node q, we run Dijkstra's algorithm on the reversed edges of the graph (Algorithm 1). Specifically, we maintain a priority queue of the current shortest distance from each node to query node q. Each time, we dequeue the node t with the shortest distance $d(t, q)$, add t to the tree by making it a child of its successor node in the shortest path from t to q, and update the distance from t's neighbors to query node q. At the same time we conduct a *rank refinement* for t; that is we compute $Rank(t, q)$. This rank refinement procedure (*GetRank* in Line 9 of Algorithm 1) will be explained shortly. During the tree construction process, every time we dequeue a node t and after updating $Rank(t, q)$, we maintain the set R of the nodes with the lowest $Rank(t, q)$ values (to be output at the end of the algorithm). The largest of the k lowest $Rank(t, q)$ values so far is denoted by $kRank$ and serves as a bound. The tree construction finishes when the shortest paths from all nodes to q have been determined.

For a large graph, constructing the entire SDS-tree is too expensive. We now show some nice properties of the SDS-tree that can help us to compute the reverse k-Ranks results, without having to build the whole tree.

LEMMA 1. *Consider a weighted graph $G = (V, E)$ and two nodes $p, q \in V$. For any node p' whose shortest path to q passes through p, Rank(p',q) $\geq$ Rank(p,q).*

PROOF. According to Definition 1, there must exist two sets S and T, such that Rank(p,q)=$|S| + 1$ and Rank(p',q)=$|T| + 1$. The aim here is to prove that $S \subset T$, which means that $|S| \leq |T|$. By definition, we know that $\forall p_i \in S$, $d(p, p_i) < d(p, q)$. Since all weights are non-negative, we further have $d(p', p) \geq 0$. Also, a path from p' to any p_i passes through p, which means that $d(p', p_i) \leq d(p', p) + d(p, p_i)$. As a result, we have $d(p', p_i) \leq d(p', p) + d(p, p_i) \leq d(p', p) + d(p, q) = d(p', q)$, therefore $\forall p_i \in S, p_i \in T$; i.e., $S \subset T$. □

Based on Lemma 1, we can easily obtain the following fundamental theorem:

THEOREM 1. *Given a SDS-tree T_q rooted at q and a node p of T_q, for any descendant p' of p, Rank(p',q) $\geq$ Rank(p,q).*

Based on Theorem 1, we can conclude that, given a SDS-tree T_q rooted at q, if node p is not in the reverse k-ranks query result of node q, then no child of p can be part of the result. This means that p's children need not be added to T_q during the tree construction; this can greatly limit the number of nodes p' that are added to the tree and for which $Rank(p', q)$ needs to be computed.

3.2 Rank Refinement

During the SDS-tree construction, for each node p that we visit and it is a candidate reverse k-ranks result, we have to apply a *rank refinement* procedure which computes $Rank(p, q)$. This is done by counting all nodes whose distance from p is shorter than $d(p, q)$. For this purpose, we build a *partial* Dijkstra tree starting from node p and we stop when we find q. The number of nodes that we encounter by this search is $Rank(p, q)$.

Recall that in Algorithm 1 we keep track of the set R of the lowest Rank values so far and of the current k-th top Rank value, denoted by $kRank$. During the refinement of $Rank(p, q)$, p can be pruned as soon as the number of nodes in the partial Dijkstra tree before reaching q is larger than $kRank - 1$, since in this case p has no potential to become a reverse k-ranks result, as well as its children nodes in the SDS-tree. The rank refinement step for a node is described by Algorithm 2.

Algorithm 2 Rank Refinement Algorithm

```
procedure GetRank($node, kRank$)
    Priority Queue $Q \leftarrow$ {node:0}                 ▷ Nodes to visit
    $D \leftarrow \varnothing$                              ▷ Nodes visited
    rank $\leftarrow 1$
    while Q do
        $top \leftarrow Q.pop()$
        $D \leftarrow D \bigcup \{top\}$
        for t in top.neighbors() do
            if t not in $D$ then
                $dis \leftarrow top.dis + d[top][t]$
                if $t \in Q$ and $t.dis > dis$ then
                    $t.dis \leftarrow dis$
                else if $t \notin Q$ and $node.dis > dis$ then
                    $t.dis \leftarrow dis$
                    $Q.push(t)$
                    $rank \leftarrow rank + 1$
                    if $rank > kRank$ then
                        return $-1$                        ▷ Definition 2
    return $rank$
```

For the example of Figure 1, the SDS-tree is the same as the Dijkstra tree since G is an undirected graph (see Figure 2). Assume that k=2. The priority queue initially has 'Alice' as top element; after that, Algorithm 1 will perform rank refinement for Bob and get $Rank(Bob, Alice) = 3$. Since Bob can be in the reverse 2-ranks results of Alice, the neighbors of Bob (i.e. Caroline and Eric) are added to the priority queue Q. Because Eric's distance to Alice is shorter than that of Caroline, we will first do rank refinement of Eric, and get $Rank(Eric, Alice) = 6$. Then, the neighbors of Eric (i.e. Sid, George and Frank) will all be added to the priority queue. Then, since Caroline has the shortest distance to Alice, we will do rank refinement for it and get $Rank(Caroline, Alice) = 4$. After that, Frank, Sid and George will be rank-refined one by one.

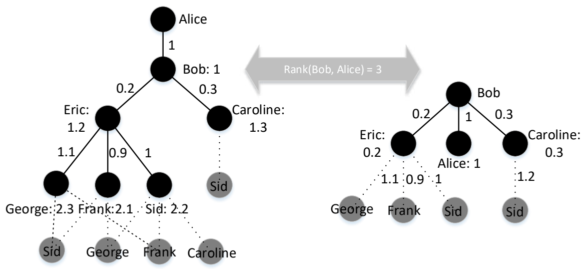


Figure 2: The SDS-tree for Example 1

4. DYNAMIC BOUNDED SDS-TREE

The filter-and-refinement framework described in the previous section significantly reduces the search space and it is much faster than the brute-force approach of computing the entire rank matrix. On the other hand, there may still be many false hits, i.e., nodes which are ranked-refined without ending up in the reverse k-ranks result. In Algorithm 1, each node is decided to be a candidate or not, immediately after refining its parent at the SDS-tree. However, at the time when a node p is dequeued, the current reverse k-ranks result (and the bound $kRank$) may have changed and it might be then possible to prune p just before its rank-refinement. We propose a Dynamic Bounded SDS-tree (DSDS-tree) approach, which is based on the idea of delaying the decision whether a node is a candidate to just before its rank refinement and on using a set of bounds to potentially prune the node.

In the DSDS-tree approach we maintain for each node p in the priority queue Q a lower bound of $Rank(p, q)$. p will be considered as a candidate right when it is dequeued; then, it is rank-refined only when its Rank lower bound is lower than the current k-th top Rank value (i.e. $kRank$). Only then p has a chance to enter the reverse k-Ranks query result. Specifically, we set the lower bound of $Rank(p, q)$ for each node p in DSDS-tree T_q rooted at node q as the maximum of the following three quantities: the depth of node p in tree T_q, the Rank value of its parent nodes, and the times visited so far during the rank-refinement of other nodes. Formally,

THEOREM 2. *Consider a DSDS-tree T_q rooted at query node q. For any node p whose depth is h and parent node is p', Rank(p,q)≥ max(h, Rank(p',q), $p.lcount$), where $p.lcount$ is the number of visited times for node p during the rank refinements of other nodes before actual refinement of node p itself.*

We can prove Theorem 2 by showing that each of the three quantities constitutes a lower bound by itself, therefore their maximum is a lower bound (the tightmost one, hence the most useful). In fact, we have already shown that $Rank(p', q)$ is a lower bound (Lemma 1). Now, we will prove the other two bounds.

We first demonstrate that $Rank(p, q)$ should not be less than the depth of node p in DSDS-tree T_q.

LEMMA 2. *Consider the DSDS-tree T_q of a query node q and suppose the depth for root is* 0. *For any node p whose depth is h, Rank(p,q)≥ h.*

PROOF. If p is at depth h, then the shortest path from p to q passes through $n = h-1$ nodes, i.e., the path is $\{p, p_1, p_2, ...p_n, q\}$. Since $\forall i \in [1, n], d(p, p_i) < d(p, q)$, we have $Rank(p, q) > n$, i.e., $Rank(p, q) \geq h$. □

We next show that $Rank(p, q)$ should not be less than the times that node p was visited during the rank-refinements of other nodes, before being refined itself.

LEMMA 3. *Consider a weighted graph $G = (V, E)$ and two nodes $p_1, p_2 \in V$, such that $d(p_1, q) \leq d(p_2, q)$. If $d(p_1, p_2) < d(p_1, q)$ holds, $d(p_2, p_1) < d(p_2, q)$ also holds.*[1]

PROOF. $d(p_2, p_1) = d(p_1, p_2) < d(p_1, q) \leq d(p_2, q)$, so $d(p_2, p_1) < d(p_2, q)$ holds. □

LEMMA 4. *Given a DSDS-tree T_q for a query node q, for any node p which has been visited during the rank refinements of other nodes for $p.lcount$ times before refinement of node p itself, Rank(p,q)≥ $p.lcount$.*

PROOF. For any node p, let T be the set of nodes which have been visited during the refinement of any node $p' \in T$ before node p itself, where $|T| = p.lcount$. This means that $d(p', q) \leq d(p, q)$ and $d(p', p) < d(p', q)$. Based on Lemma 3, we can conclude that $d(p, p') < d(p, q)$, which means that Rank(p,q)≥ $p.lcount$. □

In order to use Theorem 2, while building the *dynamic* SDS-Tree rooted at node q, we also need to maintain a priority queue of the current shortest distances from each node to query node q. Each time, we dequeue the node t with the shortest distance $d(t, q)$, we add t to the tree by making it a child of its successor node in the shortest path from t to q, and update the distance from t's neighbors to query node q if t successfully entered the current reverse k-ranks result set (as in Section 3). However, unlike the static SDS-tree, where for all nodes maintained in the priority queue we perform their rank refinement when we visit them, the dequeued nodes in the dynamic SDS-Tree are only rank-refined if their rank lower bound is smaller than the current $kRank$.

Consider the example in Figure 1. Similar to the basic framework of Section 3, the priority queue will initially have 'Alice' as root first. After dequeuing Alice and adding her neighbors in the queue, we will dequeue and rank-refine Bob to get $Rank(Bob, Alice) = 3$. Then, the neighbors of Bob (i.e. Caroline and Eric) will enter the priority queue. The rank refinement of Eric follows, giving $Rank(Eric, Alice) = 6$. Then, neighbors of Eric (i.e. Sid, George and Frank) will all enter the priority queue. Next, we will do the rank refinement of Caroline and get $Rank(Caroline, Alice) = 4$. The process can terminate here, since the lower bounds of ranks for Frank, Sid and Gorge are already larger than $kRank$. As a comparison, note that we would still have to do rank refinement for Frank, Sid and Gorge in the basic framework.

The lower-bound of Rank for each node can be dynamically updated during rank refinement steps. When meeting node t in the rank refinement of node p, we can update $t.lcount$ by adding 1, which can be done in constant time using a hash table. The whole space complexity will be $\mathcal{O}(|V|)$, but in practice we need much less space as the framework does not visit the nodes which are far from q.

5. INDEX-BASED SEARCH

In this section we propose an indexing approach which, when paired with the method presented in Section 4, can help to further reduce the cost of reverse k-ranks queries. A naive solution in this direction would be to precompute the entire rank matrix of size $|V| * |V|$. Starting from each node u, we can run a single-source shortest-path (SSSP) algorithm, i.e., build the entire Dijkstra tree, which can order all other nodes v by increasing $Rank(u, v)$ value. After computing the rank matrix, for each node v, we can sort the corresponding column of the matrix and obtain a ranked list of all

[1] This lemma holds for undirected graphs only. Therefore the count-based bound is not used in the case of directed graphs.

Algorithm 3 Build Dynamic SDS-Tree with Index Algorithm

```
procedure ReverseKRank($q, G$)
    Priority Queue $Q \leftarrow \{q : 0\}$                ▷ Nodes to visit
    $R \leftarrow$ top-$k$ in reverse_rank_dict            ▷ result so far
    $D \leftarrow \varnothing$                             ▷ Nodes visited
    $kRank \leftarrow k$-th top Rank in R
    while $Q$ do
        $top \leftarrow Q.pop()$
        $D \leftarrow D \bigcup \{top\}$
        if top $\in$ R then
            for t in top.neighbours() do
                if $t \notin D$ then
                    $dis \leftarrow top.dis + d[top][t]$
                    if $t \in Q$ and $t.dis > dis$ then
                        $t.dis \leftarrow dis$
                    else
                        $t.dis \leftarrow dis$
                        $Q.push(t)$
        LBound $\leftarrow$ max(top.height, top.parent.rank,
top.lcount, check_dic[top])
        if LBound $\geq kRank$ then
            continue
        $top.rank \leftarrow GetRank(top, kRank)$
        if $top.rank \neq -1$ then                         ▷ Theorem 1
            Update $R$
            $kRank \leftarrow$ new $k$-th rank in $R$
            for t in top.neighbours() do
                if $t \notin D$ then
                    $dis \leftarrow top.dis + d[top][t]$
                    if $t \in Q$ and $t.dis > dis$ then
                        $t.dis \leftarrow dis$
                    else
                        $t.dis \leftarrow dis$
                        $Q.push(t)$
    return $R$
```

other nodes u with respect to $Rank(u, v)$. For any reverse k-ranks query q, we can then return as results the first k nodes in the ranked list of q. The problem of this naive solution is that it is too expensive to precompute the entire rank matrix for very large graphs. Instead, we propose to only select a subset of H nodes, called *hubs*, and only run M iterations of SSSP from each hub node s to obtain s's top-M list of nearest nodes. For each node $t_i, i = \{1, 2, ..., M\}$ in this list, we simply know that $Rank(s, t_i)$ is the order of t_i in the list. The index also includes two additional components: a *Check Dictionary* and a *Reverse Rank Dictionary*, to be explained in Section 5.2. The index can facilitate the evaluation of reverse k-ranks, for k values not exceeding a parameter K. Index parameters H, M, and K are defined based on a precomputation cost/ speedup tradeoff. The larger these values are the more time it takes to create and maintain the index; on the other hand, reverse k-ranks queries are evaluated faster. In Section 6 we study the overhead and effectiveness of the index for various values of these parameters.

In the following, we first describe strategies for selecting the hub nodes and then show how we can use the precomputed information to initialize the index that can help to accelerate the computation of reverse k-ranks queries.

Algorithm 4 Dynamic Rank Refinement with Index Algorithm

```
procedure GetRank($node, kRank$)
    Priority Queue $Q \leftarrow$ {node:0}                 ▷ Nodes to visit
    $D \leftarrow \varnothing$                             ▷ Nodes visited
    rank $\leftarrow 1$
    while Q do
        $top \leftarrow Q.pop()$
        $D \leftarrow D \bigcup \{top\}$
        Update reverse_rank_dict
        for t in top.neighbours() do
            if t not in $D$ then
                $dis \leftarrow top.dis + d[top][t]$
                if $t \in Q$ and $t.dis > dis$ then
                    $t.dis \leftarrow dis$
                else if $t \notin Q$ and $node.dis > dis$ then
                    $t.dis \leftarrow dis$
                    $Q.push(t)$
                    $rank \leftarrow rank + 1$
                    $t.lcount \leftarrow t.lcount + 1$
                    if $rank > kRank$ then
                        check_dic[node] $\leftarrow D$.size()
                        return $-1$                        ▷ Definition 2
    check_dic[node] $\leftarrow rank$
    return $rank$
```

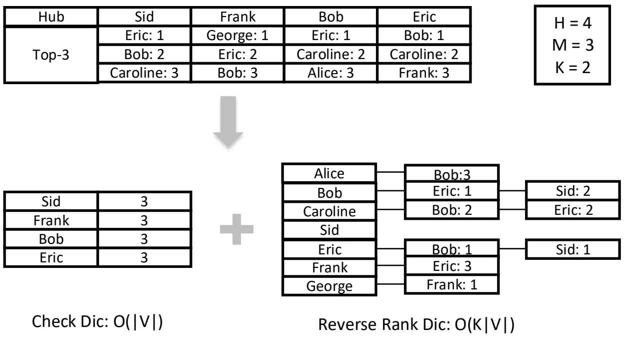
5.1 Hub Selection

We propose the following three strategies for selecting the hubs: random, degree first, closeness first. These strategies are experimentally evaluated in Section 6.

Random: We select the hubs randomly; this is used as a baseline to show the significance of other strategies.

Degree First: We select the vertices with the highest out-degrees as hubs. The reasoning behind this strategy is that vertices with higher out-degree have higher chances to connect with short shortest paths to many other vertices and therefore they have higher probability to be reverse k-Ranks query results.

Closeness First: We select as hubs the vertices with the highest closeness centrality. If we define farness of a node v as the sum of its distances from all other nodes, then closeness centrality is defined as the reciprocal of farness, which is $C(v) = 1/\sum_u d(u, v)$ [2, 18]. Since computing exact closeness centrality for all vertices requires $\mathcal{O}(|V| \cdot |E|)$ time [3] even for sparse graphs, we approximate closeness centrality by randomly sampling a small number of vertices and computing distances from those vertices to all vertices as in [1].



Hub	Sid	Frank	Bob	Eric
Top-3	Eric: 1	George: 1	Eric: 1	Bob: 1
	Bob: 2	Eric: 2	Caroline: 2	Caroline: 2
	Caroline: 3	Bob: 3	Alice: 3	Frank: 3

H = 4
M = 3
K = 2

Sid	3
Frank	3
Bob	3
Eric	3

Figure 3: Index

5.2 Index Creation

After selecting H hubs following one of the heuristic strategies mentioned above, we build the index by running the SSSP algorithm from each hub node u and stopping after obtaining the M nearest nodes from u. For each hub we store the list of M nearest nodes with their ranks. The index has two components: 1) the *Check Dictionary*; a hash-map, having nodes as keys and the number of steps SSSP has taken starting from these nodes as values (i.e., it contains an entry $\{u : M\}$, for each hub node u at the beginning); and 2) the *Reverse Rank Dictionary*; a set of adjacency lists, each corresponding to a node v and containing the current reverse K-ranks result of v ordered by rank value from small to large (recall that K is the maximum possible value of k).

Following the running example, suppose we choose {Sid, Frank, Bob, Eric} as the hubs, and precompute their top-3 rank list (i.e., $H = 4$ and $M = 3$). This gives us the rank matrix shown at the top of Figure 3. Assume that the largest possible value for k is $K = 2$. Then, we can set up the index with the following two parts. The *Check Dictionary* is {Sid:3, Frank:3, Bob:3, Eric:3} since from these four hub nodes we have so far retrieved the 3 nearest neighbors. The *Reverse Rank Dictionary* stores the existing reverse K-ranks result list for each hub node. Consider, for example, hub node Bob; we already know three Rank values, i.e. $Rank(Eric, Bob) = 1$, $Rank(Sid, Bob) = 2$ and $Rank(Frank, Bob) = 3$, but we only need to store the top-2 ranks in the Reverse Rank Dictionary, i.e., $Rank(Eric, Bob) = 1$ and $Rank(Sid, Bob) = 2$.

5.3 Querying and Index Updates

The proposed index is dynamic and can be updated whenever a new reverse k-ranks query is evaluated. For a new query node q, we first look up the *Reverse Rank Dictionary* to get any existing reverse k-ranks query results for q, which can give us an estimation for the k-th top Rank value (i.e. $kRank$). However, what we get from the index may not be the final query result; we may have to conduct more graph exploration. For this, we follow the general two-step framework described in the previous sections: we build the dynamic bounded SDS-tree, and do rank-refinement for candidate nodes. The index allows us to have a better estimation of the rank value $Rank(u, q)$ for each candidate node u: if u is in the *Reverse Rank Dictionary* of q, there is no need to do rank refinement for node u; if u is not in the *Reverse Rank Dictionary* of q, but the value of node u in the *Check Dictionary* is no smaller than the current $kRank$ value, there is also no need to do rank refinement for node u (i.e., u can be pruned). Otherwise, we have to do rank refinement for node u. For this purpose, we conduct SSSP search from u. During SSSP search, until the rank value of the nodes that we visit exceeds *Check Dictionary*$[u]$, we have to update *Reverse Rank Dictionary*. Specifically, if we reach node v with $Rank(u, v) = t_1$, and t_1 is smaller than the highest rank value of *Reverse Rank Dictionary*$[v]$, then we have to update *Reverse Rank Dictionary*$[v]$ with $\{u : t_1\}$. After finishing the rank refinement step from node u with $Rank(u, v) = t_2$, we also need to update the *Check Dictionary* with $\{u : t_2\}$. Algorithms 3 and 4 show how the search algorithm and its rank refinement module are adapted to use the index.

Following the previous example, we select {Sid, Frank, Bob, Eric} as the hubs, and precompute their top-3 ranks list as initial index. Consider Alice as the query. The index will be updated as shown in Figure 4. The index initially is as shown in Figure 3. The first step is to do rank refinement for Bob. However, as we can see in the index, the *Reverse Rank Dictionary* of Alice has {Bob:3}, which means that we need not update or compute anything and we can just turn to Eric directly. During the rank-refinement step of Eric, we also get the rank of other nodes for node Eric, and we can update the *Check Dictionary* and *Reverse Rank Dictionary* correspondingly: We add {Eric: 4} for Sid, {Eric: 5} for George, and finally {Eric: 6} for Alice, which is also the query node. After we reach Alice starting from Eric with Rank equals to 6, we also update the *Check Dictionary* with {Eric: 6}. Continuing this way, we proceed to rank-refine the next node (Caroline), and terminate, after updating again the *Check Dictionary* and the *Reverse Rank Dictionary*. Even though index updates incur extra costs (compared to the algorithm presented in Section 4), the updated index can help to speed up processing of future reverse k-ranks queries, as we demonstrate in the next section.

The space complexity for *Check Dictionary* and *Reverse Rank Dictionary* is $\mathcal{O}|V|)$ and $\mathcal{O}(K \cdot |V|)$, respectively. The overall space complexity for both index componets is $\mathcal{O}(K \cdot |V|)$. The time complexity of building the index is reduced from $\mathcal{O}(|V| \cdot (|V| + |E| \cdot \log |V|))$ of the whole matrix building to $\mathcal{O}(H \cdot (M + |E^*| \cdot \log M))$ now, where $|E^*|$ is the number of edges linked with M nodes, estimated as $|E^*| = M \cdot |E|/|V|$ and bounded by $\mathcal{O}(|E|)$.

6. EXPERIMENTAL EVALUATION

In this section, we conduct an experimental evaluation for the effectiveness of reverse k-ranks queries on large graphs and verify the efficiency of our proposed algorithms. All tested methods were implemented in C++ and the experiments were conducted on a Intel(R) Xeon(R) CPU E7-4870 @ 2.40GHz machine, with 1 TB of main memory.

6.1 Datasets

Datasets DBLP, Epinions and SF are used in our experiments. General statistics of the three datasets are shown in Table 2.

Table 2: Data Sets

	DBLP	Epinions	SF
# of Nodes	1,314,050	75,879	321,678
# of Edges	18,986,618	508,837	800,172
Average Degree	14.45	6.71	2.49

Dataset DBLP[2] contains the collaboration graph of DBLP in May 2015. There are 1,314,050 nodes and 18,986,618 edges in this dataset. Each node in the graph denotes an author and authors who have collaborated are linked by edges. The edge weight between two nodes u and v is set to 1 divided by the number of co-authored papers by u and v increased by $\log_2 deg(u) + \log_2 deg(v)$ with normalization, where $deg(u)$ is the degree of node u [17, 11]. Setting the edge weights like this can produce less ties in the result set, which is important for unambiguous ranking quality evaluation.

Dataset Epinions[3] includes an online social network from the trust based reputation system Epinions.com. There are 75,879 nodes and 508,837 edges in this directed graph. Each node is a user of the system. One user can indicate whether he/she 'trust' another user's review (i.e., whether it is useful to him/her) and one trust statement forms an edge from the declarer to the target. Edge weights are sampled from a Zipf distribution with a skewness parameter $\alpha = 2$, as in [23].

Dataset SF contains locations of stores and road network information in San Francisco bay area. There are 408 stores in this dataset, which are crawled from GeoDeg[4]. The road network was made public during the 9th DIMACS Implementation Challenge[5].

[2] http://konect.uni-koblenz.de/networks/dblp_coauthor

[3] http://snap.stanford.edu/data/soc-Epinions1.html

[4] http://geodeg.com

[5] http://www.dis.uniroma1.it/challenge9/download.shtml

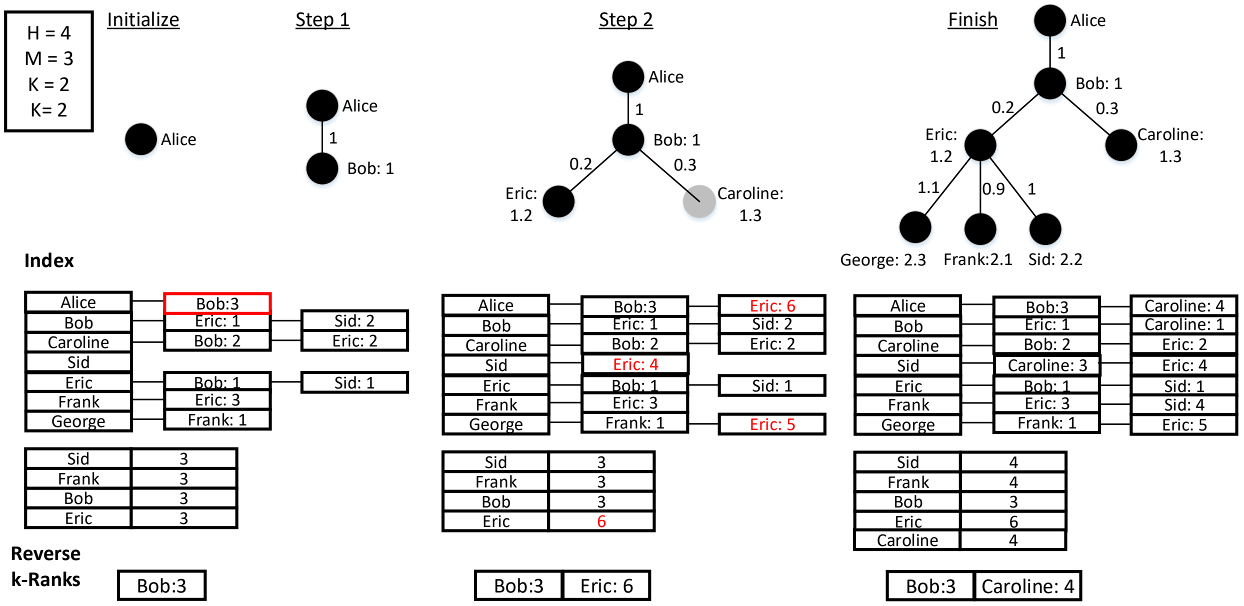


Figure 4: Index Update

There are 321,270 nodes and 800,172 edges in the network. Each store is mapped to the nearest network node. The weight on an edge (u, v) models the travel time between nodes u and v.

6.2 Effectiveness Analysis

To demonstrate the effectiveness of graph based reverse k-ranks queries, we conduct analysis at different levels of granularity that illustrate the problems of reverse top-k and top-k queries and the superiority of reverse k-ranks queries.

6.2.1 *Coarse-grained Analysis*

In our coarse-grained analysis, we investigate the results of top-k and reverse top-k queries on the DBLP dataset.

Reverse top-k query. Table 3 shows some statistics about the result sizes of reverse top-k queries on the DBLP dataset. k varies from 5 to 100. From the results, we can see that the size of result sets is not balanced. Reverse top-k queries return results of different cardinality for different query nodes. No matter what the value of k is, there always exists a large percentage of query nodes with empty reverse top-k result sets. When we increase the value of k from 5 to 100, the number of query nodes whose result set is empty decreases, but it remains in the same order of magnitude. At the same time, the largest result set size increases to 6,385, which is impractical to users.

Table 3: Reverse Top-k Result Set Size

k	5	10	20	50	100
largest set size	327	560	1,031	2,596	6,385
# of empty set	315,424	240,378	190,105	155,927	148,238
# of small set (≤ 5)	1,004,448	757,906	529,390	301,321	213,192
# of large set (≥ 100)	332	3,765	32,686	158,412	311,874

Top-k query. The problem of the top-k queries is that they are unilateral, i.e., the nodes that the query node ranks highest may not rank the query node high as well. To illustrate this problem, we investigate whether query nodes and the returned nodes have each other in their top-k results. We use $\mathbb{1}(i, j)$ to indicate whether nodes i and j agree with each other:

$$\mathbb{1}(i, j) = \begin{cases} 1, & \text{if } i \in \text{top}_k[j] \text{ and } j \in \text{top}_k[i] \\ 0, & \text{otherwise} \end{cases}$$

where $\text{top}_k[j]$ is the set of k-nearest nodes to j. Then the *agreement rate* can be calculated as:

$$\text{agreement rate} = \frac{\sum_i \sum_{j \in \text{top}_k[i]} \mathbb{1}(i, j)}{\sum_i |\text{top}_k[i]|}$$

Table 4 shows the agreement rate for various values of k. From the result, we can see that only less than half of the nodes in a top-k result also include the query node in their own top-k lists, i.e., a low agreement rate. Therefore top-k queries cannot be used as a substitute of reverse top-k and reverse k-ranks queries.

Table 4: Agreement Rate of Top-k Queries on DBLP

k Value	5	10	20	50	100
Agreement Rate(%)	48.53	44.65	41.10	37.88	35.65

6.2.2 *Fine-grained Case Study*

Wellcome and Parknshop are the two most popular supermarket chains in Hong Kong, having branches almost everywhere. We randomly choose a Wellcome and a Parknshop supermarket nearby on Google Maps, locate their nearby representative communities, and measure the road network distance between them as shown in Figure 5. In this case study, as we can see, the nearest representative community to Parknshop is B, however, if someone lives in B, he/she will prefer Wellcome to Parknshop since Wellcome is nearer. Instead, A and D would prefer Parknshop over Wellcome. Thus, the result of a top-1 query for Wellcome and Parknshop (B in both cases) is less meaningful for recommendation or advertisement compared to the result of the reverse-1 ranks query (B and A, respectively).

The reverse top-1 query here returns $\{A, D\}$ for Parknshop and $\{B, C, E, F, G\}$ for Wellcome, which is reasonable compared to

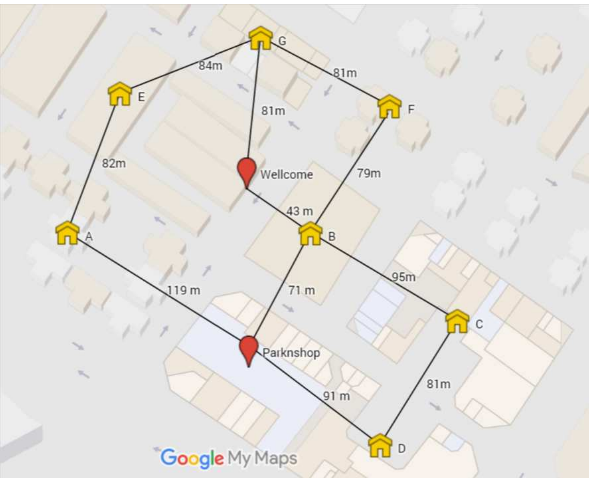


Figure 5: Case Study of Wellcome and Parknshop

Table 5: Parameters (default values in bold)

Parameter	Values
k	5,10,20,**50**,100
$h = H/\|V\|$	0.03,0.05,0.07,**0.1**,0.15
$m = M/\|V\|$	0.03,0.05,0.07,**0.1**,0.15
hub strategy	Random, **Degree First**, Closeness First

top-1 query, though with unfixed size. However, the relatively large size of results also means higher cost for promotion for the companies. On the other hand, the reverse k-ranks query defines an *ordering* of the communities with respect to their preferences on the supermarkets which can be used to prioritize market promotion to the communities that have higher chances to use it, in case of a limited budget.

6.3 Efficiency Analysis

In this section, we evaluate the performance of the reverse k-ranks approaches proposed in this paper, namely the static SDS-tree (Section 3), the dynamic SDS-tree (Section 4), and the dynamic SDS-tree with index (Section 5). We first measure the cost of the approaches as a function of different parameter values of the problem and the index. Then, we evaluate the effectiveness of the bounds used by the dynamic SDS-tree method. Next, we assess the cost of updating the index, and finally we study the efficiency of our methods on bichromatic instances of the problem.

6.3.1 Varying the Parameter Values

We evaluate the performance of our methods as a function of the following parameters: (1) size of the result set k, (2) percentage of hub nodes $h = H/|V|$, (3) percentage of ranked nodes in each index entry $m = M/|V|$, (4) hub selection strategy. Table 5 summarizes the range of values and the default value of each parameter. We measure performance by means of (i) query time and (ii) pruning power. For each setting of parameter values we run 1000 random queries and average the measures. Pruning power is measured by the average number of times the Rank Refinement function is called (we call this measure Rank Refinement in the following). The larger the Rank Refinement value is, the lower the pruning power of the method is.

Effect of k. To study the effect that the result size k has in the performance of reverse k-ranks queries, we fix the other parameters to their default values (see Table 5), and vary k from 5 to 100. As Figure 6 shows, the evaluation cost increases with k, which is consistent with the expectation that the search space and the number of candidates increases with k. Note that the dynamic approach has significantly reduced average query time for both datasets, which can be explained by the greatly reduced number of rank-refinements. The indexing method further reduces the query time to less than a few seconds, with the help of the precomputed index. We observe that the index has a greater effect on time for smaller values of k on DBLP, which can be explained by the fact that the information needed by the queries has higher chance to already be present in the index for smaller values of k. Besides, as we can see, the index works better for Epinions than for DBLP when k is large. This is because of larger average degrees in DBLP. In DBLP, even though the number of Rank Refinement calls is reduced significantly, the reduction of the average query time is not as high. This is due to the fact that the index mainly helps to avoid rank refinements that are cheap (they correspond to cases where the resulting rank is low). DBLP is a much larger and denser graph than Epinions and it is often the case that the reverse k-ranks of a query q are nodes that are quite far from q (i.e., they have low ranks). Therefore the rank refinements that are not avoided can be quite expensive, so the numerous cheap rank-refinements that are prevented due to the index have less profound effect to the query cost.

Table 6: Results with Different h on DBLP

Hub Percentage h	Index Size	Query Time (s)	Rank Refinement
0.03	1.2G	2.80015	166.702
0.05	1.2G	2.77694	151.608
0.07	1.2G	2.74801	139.514
0.1	1.2G	2.60599	124.591
0.15	1.2G	2.59796	124.591

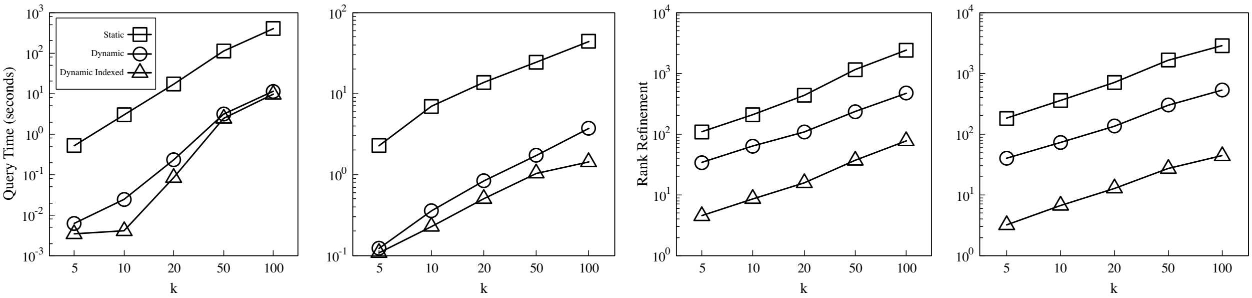
Table 7: Results with Different h on Epinions

Hub Percentage h	Index Size	Query Time (s)	Rank Refinement
0.03	25M	1.102102	70.431
0.05	27M	1.015720	66.483
0.07	29M	1.007760	63.699
0.1	30M	0.940826	59.044
0.15	32M	0.919234	51.488

In order to assess the effectiveness of our framework, we also ran tests using a naive reverse k-ranks method, described at the end of Section 2. Given the query node q, this method naively computes $Rank(p, q)$ for every node $p \in V$, by running SSSP from p until q is encountered. The top-k $Rank(p, q)$ values are tracked and eventually returned to the user. For $k = 1$, the average runtime of this naive approach on Epinions is 701.18s with 75878 Rank Refinements (For DBLP dataset, the average runtime of this naive approach is over 2000s, which is terminated by us manually), i.e., the naive method is significantly slower than the static SDS-tree approach and the dynamic SDS-tree approach with index.

Effect of hub percentage. The second parameter of which the effect we investigate is $h = H/|V|$, i.e., the percentage of hubs in the index. As shown in Tables 6 and 7, on both datasets, the average query time and the number of Rank Refinement calls decrease as h increases. Still, even for small h values, the query cost is not much higher compared to the default value. On the other hand, the index size is bounded and does not increase significantly as h increases.

Effect of index percentage. The third parameter that we study is $m = M/|V|$, i.e., the percentage of precomputed neighbors for



(a) DBLP Time (b) Epinions Time (c) DBLP Rank (d) Epinions Rank

Figure 6: Results with Different k

Table 8: Results with Different m on DBLP

Index Percentage m	Index Size	Query Time (s)	Rank Refinement
0.03	1.2G	2.756210	125.358
0.05	1.2G	2.723310	124.952
0.07	1.2G	2.626200	124.669
0.1	1.2G	2.605990	124.591
0.15	1.2G	2.577100	124.291

Table 9: Results with Different m on Epinions

Index Percentage m	Index Size	Query Time (s)	Rank Refinement
0.03	22M	0.970253	60.900
0.05	26M	0.963204	60.201
0.07	28M	0.954575	59.817
0.1	30M	0.940826	59.044
0.15	33M	0.912329	57.963

each node. As shown in Tables 8 and 9, similar to the effect of h, both the average query time and Rank Refinement calls decrease as m increases on the two real datasets. Again, the differences are not dramatic compared to the default value of m.

Effect of hub selection stategy. Next we test the effect of different hub selection strategies. As Table 10 shows, the Degree First and the Closeness First strategies are superior compared to the baseline Random strategy. Although on both DBLP and Epinions, Degree First is the winner, Closeness First performs quite similarly.

Table 10: Results with Different Hub Selection Strategies

Dataset		Random	Degree First	Closeness First
DBLP	Query Time (s)	2.861070	2.605990	2.665950
	Rank Refinement	169.500	124.591	135.132
Epinions	Query Time (s)	1.07950	0.940826	0.948437
	Rank Refinement	80.347	59.044	59.409

6.3.2 Bound Analysis

In this experiment, we test the effect of the three components of the Rank lower bound of Theorem 2. We ran 1000 random queries on Epinions. For each query and each candidate node, we count how many times each component wins as a maximum. The results are shown in Table 11. As we can see, in most cases, the rank of the parent offers the tight-most bound. In addition, although simple, height is a useful bound especially when the candidate nodes are close to the query node (i.e., if the nodes have large degrees, or when k is small). However, the effect of height declines when the value of k increases. On the other hand, the Count component is not very effective when k value is small, since only around 1% of the pruned cases are due to this bound. Besides, this bound cannot be applied for directed graphs, and it also has significant space requirements (i.e. $\mathcal{O}(|V|)$). The Count component is useful only for candidate nodes that are quite far from the query node (i.e., if the nodes have small degrees, or when k is large).

Table 11: Bound Analysis of Theorem 2

k	1	5	10	20	50	100
Height wins	87.74%	39.35%	27.48%	17.96%	9.50%	5.80%
Count wins	0.00%	0.44%	0.71%	1.07%	1.76%	2.38%
Parent wins	12.26%	60.21%	71.81%	80.97%	88.74%	91.82%

We also test the performance on Epinions by choosing 1000 queries with largest degree or fewest degree, using the dynamic SDS-tree algorithm with the four different bound strategies listed below:

- Dynamic-Parent: Rank(p,q)$\geq$ max(Rank(p',q))
- Dynamic-Count: Rank(p,q)$\geq$ max(Rank(p',q), $p.lcount$)
- Dynamic-Height: Rank(p,q)$\geq$ max(h, Rank(p',q))
- Dynamic-Three: Rank(p,q)$\geq$ max(h, Rank(p',q), $p.lcount$)[6]

The results are shown in Table 12 and Table 13. They also demonstrate that Height Component works better for nodes with large degrees while Count component works better for nodes with small degrees. As shown in Table 12, Height Component can significantly reduce Rank Refinement calls especially for small k values, while Count component brings in extra cost, which further increases query time when combining the three components compared to when using only the Height Component (an exception is the case of a large k value, i.e. k=100). On the other hand, the Count Component works better when the degree of the query node is low (i.e. Table 13) and the value of k is large.

In general, the rank of the parent offers the tight-most bound, while the Height Component helps when k is small or the query node's degree is large (i.e., candidate nodes are close to the query node). The Count Component is the least useful, being effective only when k is large or the query node has a small degree.

[6] Here we use the same symbol as in Section 4, where the depth of node p is h, node p' is the parent node of node p, and node p has been visited during the rank refinements of other nodes for $p.lcount$ times before the refinement of node p itself.

Table 12: Results with Different Bound Strategies Tested on Queries with Max Degree

k		1	5	10	20	50	100
Dynamic-Parent	Query Time (s)	0.001347	0.001348	0.001388	0.001586	0.003025	0.006705
	Rank Refinement	124.494	125.208	134.046	193.684	744.034	1738.360
Dynamic-Count	Query Time (s)	0.001385	0.001386	0.001419	0.001603	0.002872	0.006229
	Rank Refinement	124.211	124.913	133.425	189.876	690.931	1554.540
Dynamic-Height	Query Time (s)	0.000584	0.000618	0.000687	0.000856	0.001507	0.004156
	Rank Refinement	1.000	5.096	11.048	30.802	185.770	584.523
Dynamic-Three	Query Time (s)	0.000645	0.000680	0.000743	0.000917	0.001541	0.004076
	Rank Refinement	1.000	5.096	11.046	30.542	178.782	541.056

Table 13: Results with Different Bound Strategies Tested on Queries with Min Degree

k		1	5	10	20	50	100
Dynamic-Parent	Query Time (s)	0.001581	0.105760	0.258846	0.533142	1.388120	2.738640
	Rank Refinement	1.568	20.987	32.134	56.732	134.454	253.125
Dynamic-Count	Query Time (s)	0.001599	0.100026	0.269425	0.513132	1.318810	2.706620
	Rank Refinement	1.568	20.712	31.827	56.443	132.923	248.364
Dynamic-Height	Query Time (s)	0.001614	0.106397	0.271879	0.500304	1.358450	2.736410
	Rank Refinement	1.359	14.27	28.595	54.818	134.091	253.103
Dynamic-Three	Query Time (s)	0.001668	0.111674	0.257887	0.499597	1.274270	2.696320
	Rank Refinement	1.359	14.229	28.411	54.576	132.588	248.342

6.3.3 *Index Update Analysis*

In the next experiment, we evaluate the effectiveness of index updates. We randomly select 6,000 queries for each of the two real datasets and applied these queries in four different ways. We divided the 6,000 queries into n sets of the same size, i.e., for $n = 6, 3, 2, 1$, each set has 1,000, 2,000, 3,000, 6,000 queries respectively. Then, we run the Dynamic SDS-Tree with Index method n times and computed the average query time (including index update time) as well as the average number of Rank Refinement calls. All four different times apply the same 6,000 queries in the same order, the only difference being that when $n > 1$, the index is initialized (i.e., reset) multiple times. For example, when $n = 6$, the index is initialized and updated during the first 1,000 queries, then re-initialized, for the next 1,000 queries, etc. The objective is to understand whether and how the index performance improves as the index gets updated. Table 14 shows the average runtime and conducted rank-refinements per query. We can see that the more the index evolves the more rank refinements can be avoided and the more the average query time decreases.

Table 14: Results with Index Update

Dataset		Query Time (s)	Rank Refinement
DBLP	1,000	2.6287438	130.255
	2,000	2.530356	126.634
	3,000	2.486031	123.263
	6,000	2.228565	115.641
Epinions	1,000	1.179105	61.407
	2,000	0.985524	50.599
	3,000	0.924317	45.206
	6,000	0.544288	34.958

Table 15 shows the cost for initializing the index for various values of h and m. Although the times are quite high (especially for large values of h and m), this one-time cost pays off, because the index can help to achieve substantial cost savings for reverse k-ranks queries, as already shown.

Table 15: Index Construction Time (hours)

h	m	DBLP	Epinions
0.03	0.1	2.68	0.01
0.05	0.1	4.08	0.02
0.07	0.1	6.21	0.03
0.1	0.1	8.94	0.04
0.15	0.1	12.94	0.06
0.1	0.03	2.95	0.01
0.1	0.05	3.92	0.02
0.1	0.07	6.40	0.03
0.1	0.1	8.94	0.04
0.1	0.15	12.79	0.06

6.3.4 *Bichromatic Queries*

Although our solutions are only described in a monochromatic context, they are readily available for the case where the graph nodes are divided into two classes and the nodes are ranked with respect to where they are ranked by nodes that belong to the other class. Examples of bichromatic top-k, reverse top-k and reverse k-ranks queries have been shown at the case study of Section 6.2.2. In this section, we evaluate the performance of our methods for bichromatic reverse k-ranks queries. For the sake of completeness, we first provide a problem definition.

DEFINITION 3. *(Bichromatic $Rank(s,t)$) Consider a bichromatic weighted graph $G = (V, E)$, consisting of a set of nodes $V = V_1 \cup V_2$ and a set of edges E. Each edge in E carries a non-negative weight. For any two nodes $s \in V_1$, $t \in V_2$, let $d(s,t)$ denote the shortest path distance from s to t, which is defined by summing up the weights of the edges along the shortest path from s to t. Let $S \subset V_2$ be the set of nodes that satisfy $\forall p_i \in S, d(s,p_i) < d(s,t)$ and $\forall p_j \in (V_2 - S - \{t\}), d(s,p_j) \geq d(s,t)$. Then, $Rank(s,t) = |S| + 1$ where $S \subset V_2$ and $|S|$ is the cardinality of S.*

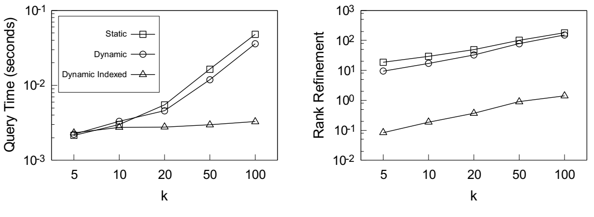
DEFINITION 4. *(Reverse k-Ranks Query on a Bichromatic Graph)*

Given a bichromatic weighted graph $G = (V, E)$, where $V = V_1 \cup V_2$, a query node $q \in V_2$ and a positive integer k, the reverse k-ranks query on a bichromatic graph returns a subset T of V_1, such that $|T| = k$ and $\forall p_i \in T$, $\forall p_j \in (V_1 - T)$, $Rank(p_i, q) \leq Rank(p_j, q)$.

In a reverse k-ranks query on a bichromatic graph, the query node is of one type, while the returned results are of another type. All our proposed methods can be used here with little modification: only for the nodes of the same type as the result set we need to do rank refinement, and only the nodes of the same type as the query node need to be counted during rank refinement. This is consistent with our case study with the communities and supermarkets; i.e., the management of a supermarket may use a reverse k-ranks query to find out which k communities the supermarket has higher chances to attract.

We use the SF road network (described in Section 6.1) to test the efficiency of our three proposed methods. We extract from the graph the nodes which are the nearest ones to 408 real stores and mark them as store nodes, while all other nodes are considered to be community nodes. We vary k from 5 to 100 in the experiment and measure the average query time and the average number of rank refinements per query, as performance metrics. The results are plotted in Figure 7. As we can see, when k is small, even though the Dynamic and Dynamic-Indexed methods can reduce the number of rank refinements, their average query time is not reduced; the cost of these methods is higher than the static approach for k=5. This is because the overhead cost by the data structures maintained by the Dynamic and Dynamic-Indexed methods. However, for larger k values, the superiority of the dynamic approaches and the index becomes apparent. Note that in this case of a sparse graph, the index approach is much more efficient compared to the static/dynamic SDS-tree method without index.

The general conclusions from our experimental study are as follows: (1) The reverse k-ranks query produces more useful results compared to the top-k query and the reverse top-k in recommendation applications, where a ranked set of objects of certain size needs to be recommended to the query object q; (2) Our filter-and-refinement framework is very efficient compared to the naive approach; (3) In the dynamic SDS-tree method, the parent-based and height-based bounds have higher applicability and effectiveness compared to the count-based bound, but count-based bound also has its applicable scenario; (4) The index is more effective for sparse graphs and for medium to high values of k.



(a) SF Time

(b) SF Rank

Figure 7: Performance on a Bichromatic Network

7. RELATED WORK

Ranking queries have been widely used in many applications and have many variants. The most general one is the top-k query which returns the k objects with the highest scores based on a ranking function. Recently, reverse ranking queries emerged, which rank from the perspective of result objects, not the query objects.

Top-k query. Top-k queries have already been studied extensively for decades [10]. They return a list of objects which are ranked using an aggregate function that applies on their features. The most famous algorithms for top-k queries are Fagin's Threshold Algorithm (TA) and No Random Accesses (NRA) [5]. They are designed for rank-combining sorted lists of objects based on different attributes (features). TA allows both sequential and random accesses to these lists, whereas NRA allows only sequential accesses. Both algorithms use bounds for the top-k results, based on the information accessed so far and terminate as soon as the current top-k results are guaranteed to be the final ones, aiming at minimizing the accesses to the input lists. Mamoulis et al. [15] proposed an improved version of NRA, which is designed to minimize the number of object accesses, the computational cost, and the memory requirements of top-k search with monotone aggregate functions. If the ranking function is defined based on the distance of the objects to a pivot object, top-k queries are referred to as k nearest neighbor (k-NN) queries. k-NN queries have extensively been studied in spatial databases [9]. The single-source shortest path (SSSP) algorithm (a simple adaptation of Dijkstra's algorithm) can be used to evaluate k-NN queries on graphs.

Reverse k-NN query. The reverse k nearest neighbor (RKNN) query returns a set of query objects that have a given query point as one of their k nearest neighbors [24]. Preprocessing-based methods usually leverage index structures for efficient RKNN query evaluation. For example, the R-tree [8] is used for RKNN queries on spatial data [13, 20]. Yiu et al. [25] studied RKNN on large graphs using shortest path as the proximity measure. Similar to the reverse top-k query, which we review next, the result size of RKNN is not fixed, which may limit its application.

Reverse top-k query. The reverse top-k query returns the aggregate functions which rank a given query object highest. Vlachou et al. [21] presented an efficient threshold-based algorithm that eliminates candidates, without having to evaluate any top-k queries using the result functions. Furthermore, they introduced an indexing structure based on materialized reverse top-k views in order to speed up the computation of reverse top-k queries. These techniques were improved later in [22]. Ge et al. [6] proposed methods that compute all top-k queries in batch by applying the block indexed nested loops paradigm and a view-based algorithm. Yu et al. [26] studied reverse top-k queries when applied on large graphs, using Random Walk with Restart distance between nodes as the ranking function. Essentially, such reverse top-k queries on graphs are equivalent to RKNN on graphs, but using a different distance measure. As explained in [27], reverse top-k (and RKNN) queries only give results for query objects which are 'hot' (i.e., easily reachable by many other nodes), while most 'cold' objects get empty or too small result sets.

Reverse k-ranks query. To solve the aforementioned problem of RKNN queries, Zhang et al. [27] propose a new ranking query: the reverse k-ranks query in vector spaces. The reverse k-ranks query returns the k objects with the smallest Rank(w,q) values, where Rank(w,q) denotes the number of objects ranking higher than q for the same ranking function w. As opposed to reverse top-k and RKNN queries, the result set size of a reverse k-ranks query is fixed.

8. CONCLUSION

This paper is the first-time ever study of reverse k-ranks queries over large graphs. We have shown through real-life case studies that reverse top-k queries may produce unsatisfactory results; therefore, there is a need for the efficient support of reverse k-ranks queries. Then, we proposed a filter-and-refinement framework for evaluating reverse k-ranks queries, based on the construction of a SDS-tree and the dynamic refinement of its nodes. We also proposed an indexing technique that can further improve the performance of the framework. Our experimental evaluation which uses three real large-scale graphs of different characteristics confirms the efficiency of the proposed techniques. In the future, we plan to study reverse k-ranks queries for other node similarity measures (i.e. PageRank, Personalized PageRank and SimRank), which require radically different approaches.

Acknowledgements

This work was supported by grants GRF 17205015 and 17201414 from Hong Kong RGC.

9. REFERENCES

[1] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*, pages 349–360. ACM, 2013.

[2] A. Bavelas. Communication patterns in task-oriented groups. *Journal of the acoustical society of America*, 22:725–730, 1950.

[3] U. Brandes and C. Pich. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, 17(07):2303–2318, 2007.

[4] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[5] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[6] 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.

[7] Z. Guan, J. Bu, Q. Mei, C. Chen, and C. Wang. Personalized tag recommendation using graph-based ranking on multi-type interrelated objects. In *SIGIR*, pages 540–547. ACM, 2009.

[8] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57. ACM Press, 1984.

[9] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[10] 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.

[11] M. Jiang, A. W. Fu, and R. C. Wong. Exact top-k nearest keyword search in large networks. In *SIGMOD*, pages 393–404. ACM, 2015.

[12] KONECT. DBLP network dataset. http://konect.uni-koblenz.de/networks/dblp_coauthor, 2015.

[13] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *SIGMOD*, pages 201–212. ACM, 2000.

[14] S. Lo and C. Lin. WMR–A graph-based algorithm for friend recommendation. In *WI*, pages 121–128. IEEE Computer Society, 2006.

[15] N. Mamoulis, M. L. Yiu, K. H. Cheng, and D. W. Cheung. Efficient top-*k* aggregation of ranked inputs. *ACM Trans. Database Syst.*, 32(3):19, 2007.

[16] F. Meng, D. Gao, W. Li, X. Sun, and Y. Hou. A unified graph model for personalized query-oriented reference paper recommendation. In *CIKM*, pages 1509–1512. ACM, 2013.

[17] M. Qiao, L. Qin, H. Cheng, J. X. Yu, and W. Tian. Top-k nearest keyword search on large graphs. *PVLDB*, 6(10):901–912, 2013.

[18] G. Sabidussi. The centrality index of a graph. *Psychometrika*, 31(4):581–603, 1966.

[19] P. Symeonidis, E. Tiakas, and Y. Manolopoulos. Product recommendation and rating prediction based on multi-modal social networks. In *RecSys*, pages 61–68. ACM, 2011.

[20] Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *PVLDB*, pages 744–755. Morgan Kaufmann, 2004.

[21] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *ICDE*, pages 365–376. IEEE Computer Society, 2010.

[22] A. Vlachou, C. Doulkeridis, K. Nørvåg, and Y. Kotidis. Branch-and-bound algorithm for reverse top-k queries. In *SIGMOD*, pages 481–492. ACM, 2013.

[23] X. Xiao, B. Yao, and F. Li. Optimal location queries in road network databases. In *ICDE*, pages 804–815. IEEE Computer Society, 2011.

[24] S. Yang, M. A. Cheema, X. Lin, and W. Wang. Reverse k nearest neighbors query processing: Experiments and analysis. *PVLDB*, 8(5):605–616, 2015.

[25] M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao. Reverse nearest neighbors in large graphs. *TKDE*, 18(4):540–553, 2006.

[26] A. W. Yu, N. Mamoulis, and H. Su. Reverse top-k search using random walk with restart. *PVLDB*, 7(5):401–412, 2014.

[27] Z. Zhang, C. Jin, and Q. Kang. Reverse k-ranks query. *PVLDB*, 7(10):785–796, 2014.

Analytics on Fast Data: Main-Memory Database Systems versus Modern Streaming Systems

[Experiments and Analyses]

Andreas Kipf[1] Varun Pandey[1] Jan Böttcher[1]
Lucas Braun[2] Thomas Neumann[1] Alfons Kemper[1]

[1]Technical University of Munich
{kipf, pandey, boettche, neumann, kemper}@in.tum.de

[2]ETH Zurich
braunl@inf.ethz.ch

ABSTRACT

Today's streaming applications demand increasingly high event throughput rates and are often subject to strict latency constraints. To allow for more complex workloads, such as window-based aggregations, streaming systems need to support *stateful* event processing. This introduces new challenges for streaming engines as the state needs to be maintained in a consistent and durable manner and simultaneously accessed by complex queries for real-time analytics.

Modern streaming systems, such as Apache Flink, do not allow for efficiently exposing the state to analytical queries. Thus, data engineers are forced to keep the state in external data stores, which significantly increases the latencies until events are visible to analytical queries. Proprietary solutions have been created to meet data freshness constraints. These solutions are expensive, error-prone, and difficult to maintain. Main-memory database systems, such as HyPer, achieve extremely low query response times while maintaining high update rates, which makes them well-suited for analytical streaming workloads. In this paper, we identify potential extensions to database systems to match the performance and usability of streaming systems.

CCS Concepts

•**Information systems** → **Stream management;**

Keywords

stream processing; main-memory database systems

1. INTRODUCTION

Gartner recently forecasted that there will be more than *20 billion* connected devices in 2020, a 400% increase compared to 2016[1]. The growing popularity of Internet of Things applications [11], including connected vehicles, cell phones, and health monitoring devices, enable a variety of new business use cases and applications. These applications are typically built around streaming systems that are able to ingest and aggregate enormous amounts of events from different data sources. Given the spike of interest in building such applications, it is not surprising that dedicated stream processing systems, like Apache Storm[2], Apache Spark Streaming[3], or Apache Flink[4], are receiving significant attention not only in the database but also in the data science and in the open-source community.

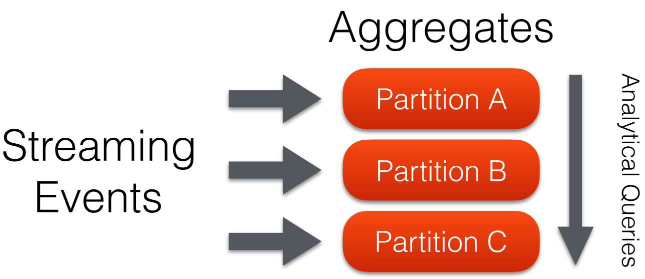


Figure 1: Analytics on fast data. Streaming events may be processed concurrently in different partitions, whereas analytical queries cross partition boundaries and require a consistent state.

To better understand the different types of workloads that these systems need to handle, we will walk through different ways of processing sensor readings (events) of connected vehicles that contain information about street conditions such as icy road segments.

First, a streaming system could warn vehicles about icy road segments based on the information of single events. In that case, the streaming system does not need to maintain state. We refer to such workloads as *stateless* streaming.

Second, a system could process the aggregated information of multiple events to decide if vehicles should be warned. Such an implementation requires the system to maintain a processing state, which introduces new challenges such as consistency and durability. We call this kind of workloads *stateful* streaming.

[1] http://www.gartner.com/newsroom/id/3165317

[2] http://storm.apache.org/

[3] http://spark.apache.org/streaming/

[4] http://flink.apache.org/

Series ISSN: 2367-2005

10.5441/002/edbt.2017.06

Third, a streaming system allows users to perform analytical queries on the entire set of aggregates (the conditions of all road segments across the city) to find the most critical segments. We refer to such workloads as *analytics on fast data*. These workloads are particularly challenging for a streaming system since it needs to perform computations across multiple partitions in a consistent manner to answer analytical queries (cf., Figure 1). In fact, without modifications, none of the streaming systems mentioned above can handle this use case.

One idea to mitigate this problem is to make use of the fact that these systems periodically flush their state to durable storage (e.g., HDFS) to address fault tolerance. This means that the system state becomes queryable for an analytical engine like Apache Spark[5]. However, the delay that this design introduces prohibits analytical queries to run on the most recent state, which is required by use cases like the one above.

Another example is the Huawei-AIM telecommunication workload described in [2]. In this use case, events represent sales and marketing information generated by phone calls. On the one hand, the application needs to maintain a huge set of aggregates per customer in order to trigger alerts for this particular customer (a *stateful* streaming workload). On the other hand, maintenance specialists might query the overall system state to localize sources for network failures or business analysts might run analytics to gather insights and propose new offers in real time (*analytics on fast data*).

Again, using an off-the-shelf stream processor does not solve the case described in [2] because it cannot handle the real-time analytics. There are, however, state-of-the-art main-memory database systems (MMDBs) dedicated to handle mixed OLTP and OLAP workloads, such as HyPer [7] and Tell[6], which seem promising because stream processing could also be seen as a particular class of OLTP workloads. These systems feature advanced query optimizers, compile queries to native code, and can thus achieve extremely low response times for complex analytical queries. Using efficient snapshotting mechanisms, such as *copy-on-write*, *MVCC*, or *differential updates* [7, 15, 8], these systems are able to sustain high transaction throughput rates in parallel to analytical query processing making them well-suited for workloads where analytical queries need to consider recently ingested data.

Despite all these advantages, it seems that data engineers are still reluctant to use MMDBs for stream processing. They either build their own solutions on top of modern streaming systems (e.g., Apache Flink) or hand-craft systems from scratch that are specifically tuned for particular workloads (e.g., AIM [2]). One reason that MMDBs are not widely used for streaming workloads is that they lack out-of-the-box streaming functionality, such as window functions, and adding this functionality (e.g., through stored procedures or user-defined functions) results in additional engineering. If MMDBs would offer a better support for streaming workloads (e.g., streaming extensions for SQL as proposed in StreamSQL [16]), they would be preferable over hand-crafted systems, which are also costly to maintain.

In this *Experiments and Analyses* paper, we thoroughly evaluate the usability and performance of MMDBs, modern streaming systems, and AIM, a hand-crafted system, using the Huawei-AIM workload [2]. Based on the evaluation results, we answer the question how off-the-shelf MMDBs can be extended to sufficiently satisfy the requirements of *analytics on fast data*. We identify a set of modifications that, if properly applied to the off-the-shelf MMDBs, allow these systems to address the needs of *analytics on fast data*.

Our contributions include:

- A rich survey of various MMDBs, modern streaming systems, and a hand-crafted system specifically designed to address the Huawei-AIM workload
- A thorough usability and performance evaluation including at least one representative of each of these classes of systems
- A discussion of how MMDBs can be extended to match the performance and usability of modern streaming systems

The remainder of this paper is structured as follows: Section 2 summarizes a broad variety of existing systems and Section 3 revisits the Huawei-AIM workload and describes how it can be implemented with these systems. Section 4 evaluates the performance of representatives of each kind of system with respect to this workload. Section 5 enumerates ideas regarding how to bridge the performance and usability gap between MMDBs and modern streaming systems and is followed by the conclusions to our evaluation presented in Section 6.

2. APPROACHES

There are numerous systems that can be used to build stream processing pipelines, including near real-time data warehousing solutions like Mesa [6] and in-memory incremental analytical engines like Trill [4]. S-Store [12] is an approach to integrating stream processing into an OLTP engine. Since addressing all of these systems is beyond the scope of this paper, we will focus on representative MMDBs, popular streaming systems from the open-source domain, and AIM [2], a hand-crafted highly-optimized solution.

2.1 Main-Memory Database Systems

There are multiple MMDBs that can handle *analytics on fast data* or more generally hybrid transactional/analytical processing (HTAP) workloads. In HTAP, transactions are usually more complex (e.g., TPC-C transactions) than the single-row transactions studied in this work.

2.1.1 HyPer

HyPer[7] is a MMDB that achieves an outstanding performance for both OLTP and OLAP workloads, even when they operate simultaneously on the same database. HyPer uses two different snapshotting mechanisms to avoid expensive synchronization. By leveraging the *copy-on-write* feature of the MMU, the *fork* mechanism [7] efficiently creates consistent copies of the database to enable analytical queries to run without interruptions. The second snapshotting mechanism [15] is based on *multi version concurrency*

[5] http://spark.apache.org/

[6] http://www.tell-project.org/

[7] When saying HyPer, we are referring to the research version of HyPer developed at the Technical University of Munich.

control (*MVCC*) and isolates transactions by versioning individual attributes. Currently, HyPer does not yet implement physical *MVCC* meaning that transactions do not run simultaneously with analytical queries but are interleaved. HyPer further features data-centric LLVM code generation with just-in-time compilation. Finally, HyPer has an advanced dynamic programming-based optimizer including the ability to unnest arbitrary queries.

2.1.2 MemSQL

MemSQL[8] is a MMDB that uses LLVM for code generation. In-memory data is organized row wise while on-disk data is organized column wise. MemSQL currently does not support stored procedures, thus making it difficult to implement stream processing workloads requiring a complex logic for updating state. To implement such a workload in MemSQL, one needs to implement the update logic externally, leading to costly round trips between the application and the database. Another alternative to implement such workloads in MemSQL is to use MemSQL Streamliner[9], which offers a connector between Spark (Streaming) and the relational database. Streaming results from Spark Streaming can be materialized into MemSQL for further investigation. The main drawback of this solution is that the two systems remain separated causing higher than necessary latencies. Further, streams cannot be joined with regular tables residing in MemSQL without materializing and transferring them to the relational database.

2.1.3 Tell

Tell is a distributed shared-data MMDB that supports OLTP and OLAP in parallel and is developed at the Systems Group at ETH Zurich. The implementation of Tell is fundamentally different from that of other systems presented in this paper as it separates the computation from the storage layer in such a way that both layers can scale out individually [10].

The storage layer, TellStore, is a versioned key-value store with additional support for fast scans and different storage layout options, such as *RowStore* and *ColumnMap. ColumnMap*, the preferred layout for HTAP workloads, was created as part of Analytics in Motion (AIM) [2] (cf., Section 2.3) and is a modified *Partition Attributes Across* (PAX) [1] approach that optimizes cache locality by storing data columnwise in blocks of cache size. This optimization allows *ColumnMap* to support fast scans and, at the same time, reasonably fast record lookups and updates. TellStore employs the *shared scan* technique, which allows incoming scan requests to be batched and processed all at once by a single thread. The *shared scan* can be parallelized efficiently by partitioning the data and using a dedicated scan thread for each of these partitions in parallel [18]. Isolation is guaranteed using a combination of *differential updates* [8] and *MVCC*. Updates are put into a *delta* data structure, which gets periodically merged with the *main* data structure that serves analytical queries. This approach is also used in SAP HANA [5].

Tell's compute layer offers two processing APIs: TellDB (C++) for general-purpose transactions and TellJava (Java) for read-only analytics. TellJava can be further integrated into distributed processing frameworks, including Apache Spark and Presto.

2.2 Modern Streaming Systems

In addition to MMDBs, there are dedicated streaming systems allowing for the implementation of streaming pipelines. These systems provide out-of-the-box functionality, including a rich set of operators to help data engineers to address the specific demands of streaming use cases.

2.2.1 Apache Samza

Apache Samza[10] is a distributed framework for continuous real-time data processing that is lightweight, elastic, and fault-tolerant. Samza uses Apache Kafka[11] (a durable publish-subscribe-based message passing system that allows replaying messages) for real-time feeds and produces output feeds for Kafka to consume. For distributed scheduling, fault tolerance, and resource allocation, Samza depends on Apache YARN and on Kafka. Samza employs a checkpointing mechanism to provide *at-least-once* guarantees. It creates checkpoints at predefined time intervals and in case of a job failure, it replays messages from the last checkpoint. A drawback of Samza is that it does not support *exactly-once* semantics. A message might be processed twice after a job failure, which can lead to non-exact results. That effect can be minimized by using shorter checkpoint time intervals.

2.2.2 Apache Flink

Apache Flink [3] is a combined batch and streaming processing system that supports *exactly-once* semantics. Flink follows a *tuple-at-a-time* approach, providing low latency. Using asynchronous checkpointing, Flink is able to decouple its fault-tolerance mechanism from the tuple processing. The processing continues while Flink periodically creates snapshots of the operator states and the in-flight tuples. Flink can achieve superior throughput compared to Apache Storm (cf., Section 2.2.4). In contrast to the other streaming systems, Flink allows for event time semantics. Flink allows the extraction of the actual event timestamp (i.e., the time when the event was originally captured) when an event arrives at the streaming engine to assign it to its appropriate window. A drawback of Flink is that current versions only allow maintaining state on an operator level. However, there is a pull request for a queryable state[12] to be released with Flink 1.2.0. The idea is to maintain an operator-independent state within Flink and expose it to external queries. Internally, the state is partitioned and guarantees fault tolerance (i.e., *exactly-once* semantics). A restriction of this solution is that it is only a key-value state supporting only point lookups. More complex queries, including full table scans, are not possible.

As a workaround, one can implement a custom operator that holds both the state and the logic for the corresponding analytical queries. The drawback of this approach is that the whole state and query logic has to be implemented manually. Further, this approach does not support concurrent stream and query processing since analytical queries can only be ingested through the stream processing pipeline itself resulting in an interleaved execution.

[8] http://www.memsql.com/

[9] http://blog.memsql.com/spark-streamliner/

[10] http://samza.apache.org/

[11] https://kafka.apache.org/

[12] https://issues.apache.org/jira/browse/FLINK-3779

2.2.3 Apache Spark Streaming

Apache Spark Streaming [19] is the streaming extension to the cluster computing platform Apache Spark. Spark Streaming organizes incoming streaming tuples into *micro-batches* that are being processed atomically thus optimizing for throughput. This approach allows the use of the same programming model for batch and stream processing. Spark Streaming supports *exactly-once* semantics.

2.2.4 Apache Storm

Apache Storm [17] is a widely used stream processing system that does not guarantee state consistency and follows a *tuple-at-a-time* approach, thus favoring low latency over throughput. Storm implements *at-least-once* semantics by keeping upstream backups of data that are being replayed if no acknowledgements have been received from downstream nodes. Trident[13] extends Storm with *exactly-once* semantics and allows running queries on consistent state.

2.3 AIM

In collaboration with Huawei, researchers of the Systems Group at ETH Zurich designed the AIM system to address the specific characteristics of a telecommunications workload. AIM is a research prototype that allows efficient aggregation of high-throughput data streams. It was specifically designed to address the Huawei-AIM workload that we use for evaluation purposes in this paper (cf., Section 3). Due to its hand-optimized nature, AIM achieves an outstanding performance on that workload and therefore serves as a baseline for our experiments. AIM has a three-tier architecture consisting of storage, event stream processing (ESP), and real-time analytics (RTA) nodes (or threads if deployed in a standalone setting). RTA nodes push analytical queries down to the storage nodes, merge the partial results, and finally deliver the results to the client. ESP nodes process the incoming event stream, evaluate alert triggers, and update corresponding records by sending *Get* and *Put* requests to the storage nodes. The storage nodes store horizontally-partitioned data in a *ColumnMap* layout and employ *shared scans* as described in Section 2.1.3. AIM can also be deployed standalone, which eliminates network costs and therefore tests the pure read, write, and scan performance of the server.

2.4 Summary

A comparison of different aspects of stream processing approaches is presented in Table 1. These aspects include:

Semantics Streaming engines make different guarantees regarding how messages (i.e., events) are being processed. A streaming engine only ensures completely correct results when providing *exactly-once* guarantees. Some engines optimize for low latency and thus often cannot provide *exactly-once* guarantees as this would require them to implement transactions, which are expensive in a distributed setting. Therefore, streaming engines often fall back to *at-least-once* semantics (i.e., a message will be resent until it is processed at least once), which are good enough for many applications. Many stream processing engines require a durable data source for *exactly-once* guarantees because they only persist their processing state at certain points of time (often called checkpoints). In case of a failure, messages need to be replayed from the last checkpoint. In contrast, database systems achieve durability through the use of redo logs and thus only need to replay messages sent during the time the database system was down. The third processing guarantee is *at-most-once*. In an *at-most-once* setting, messages might get lost but are never processed twice or more often. Few systems implement this approach since loosing data is an undesirable property for most applications.

Durability Durability is closely related to the semantics offered by stream processing systems. While some systems require a durable data source to achieve durability, others provide durability out-of-the-box.

Latency Especially in real-time scenarios, low latencies are crucial to deliver valuable results. As stated above, latency often depends on the processing guarantee offered by a system. MMDBs that often run on a single machine or are optimized for low-latency networks can yield low latencies while providing *exactly-once* processing guarantees.

Computation model There are two computation models: *tuple-at-a-time* and *micro-batch*. The natural approach is to process streams continuously. However, streams can also be batched and processed as small chunks of data. Spark Streaming follows this approach allowing it to achieve high throughput rates. However, following a *tuple-at-a-time*-based approach does not necessarily lead to lower throughput since the computation model can be independent from the checkpointing interval. For instance, Flink follows a *tuple-at-a-time*-based approach combined with a batch-based checkpointing mechanism thus optimizing for both latency and throughput. MMDBs usually treat stream events as transactions, which might also be batched for better performance (e.g., Tell processes 100 events within a single transaction).

Throughput Another important aspect in stream processing is throughput. Particularly when costs matter, higher throughput helps to reduce the number of required resources. Due to the low costs to process single-row transactions (updating aggregates of single entities), throughput mainly depends on the employed fault-tolerance mechanism and whether a system batches transactions. Throughput increases with longer checkpointing intervals.

State management For mixed OLTP and OLAP workloads, the state updated by the OLTP subsystem needs to be exposed to the OLAP subsystem. Traditional streaming engines, such as Apache Storm, do not allow maintaining state. They are only designed to process and transform an input into an output data stream preventing writing *stateful* stream processing applications (e.g., aggregations over windows). Trident extends Storm with state management capabilities. Flink only maintains states on an operator basis and currently does not support global states that can be accessed by analytical queries. Database systems, on the other hand, can persist streaming results in temporary

[13] http://storm.apache.org/documentation/Trident-state

	MMDBs			Modern Streaming Systems				
Aspect	**HyPer**	**MemSQL**	**Tell**	**Samza**	**Flink**	**Spark Streaming**	**Storm**	**AIM**
Semantics	Exactly-once	Exactly-once	Exactly-once	At-least-once	Exactly-once	Exactly-once	Exactly-once	Exactly-once
Durability	Yes	Yes	No	With durable data source	With durable data source	With durable data source	With durable data source	No
Latency	Low	Low	Low	High (writes messages to disk)	Low	Medium (depends on batch size)	Low	Low
Computation model	Tuple-at-a-time	Tuple-at-a-time	Tuple-at-a-time	Tuple-at-a-time	Tuple-at-a-time	Micro-batch	Micro-batch	Tuple-at-a-time
Throughput	High	High	High	High	High	Medium (depends on batch size)	Low	High
State management	Yes	Yes	Yes	Yes (durable K/V store)	Yes	Yes (writes into storage)	Yes	Yes
Parallel read/write access to state	Copy on write, MVCC	No	Differential updates, MVCC	No	No	No	No	Differential updates
Implementation languages	C++, LLVM	C++, LLVM	C++, LLVM	Java, Scala	Java	Java, Scala	Java, Clojure	C++
User-facing languages	SQL	SQL	C++, Java, Scala (through Spark shell), SQL (through Presto shell)	Java, Scala	Java, Scala	Java, Scala, Python, SparkSQL	Any (through Apache Thrift)	C++
Own memory management	Yes	Yes	Yes (w/ GC)	No	Yes	Yes	No	Yes
Window support	Using stored procedures	Only manually	Only manually	Very basic	Very powerful	Basic	Basic	Using template code

Table 1: Comparison of different stream processing approaches

tables allowing OLAP queries to access them as if they were regular database tables.

Parallel read/write access to state As mentioned earlier, Trident extends Storm with state management functionalities; however, it does not allow analytical queries and updates to access state in parallel. Instead, they have to be interleaved to ensure a consistent view of the state. In contrast, modern MMDBs can efficiently expose their current state to analytical queries through the use of snapshotting mechanisms, such as *copy-on-write*, *MVCC*, or *differential updates*.

Implementation languages Most of the streaming systems are written in a JVM-based language, whereas MMDBs are usually implemented in C or C++. The trend is to compile queries to native code. HyPer, Tell, and MemSQL use LLVM as a compiler backend.

User-facing languages The Apache systems support primarily JVM-based languages while the MMDBs all support SQL and, in the case of Tell, additional languages through its Spark and Presto integration.

Own memory management Whether a system employs its own memory management or fully relies on the memory management of the JVM. Spark Streaming and Flink are based on the JVM but still employ their own memory management to have a better control over garbage collection cycles.

Window support In streaming applications, aggregations are usually computed on a window basis. Two basic window types are sliding and tumbling. Sliding windows are contiguous time or count-based intervals, such as *last 24 hours* or *last 10,000 events*. Tumbling windows are non-overlapping time or count-based intervals, such as *today* or *every 10,000 events*. All of the analyzed streaming engines support these two kinds of windows. In particular, Flink offers extensive functionality to specify windows, supporting custom window assigners, triggers, and evictors. AIM supports tumbling windows for specific time intervals and the standard aggregation functions through templated code. The window definitions are loaded at startup and cannot be changed afterwards. The analyzed MMDBs have no natural window support. However, in the case of HyPer, windows can be manually implemented using stored procedures.

3. WORKLOAD

AIM was motivated by a telecommunication workload, which we will refer to as Huawei-AIM use case [2]. We chose this workload as it is well-defined and represents the workload class of *analytics on fast data*.

3.1 Description

The Huawei-AIM use case requires events, more specifically *call records*, to be aggregated and made available to analytical queries. The system's state, which AIM calls the

subscriber ID	international calls						...
	today					...	...
	count	duration			...	...	...
		sum	min	max	...	...	...

Table 2: Schema snippet of the Analytics Matrix

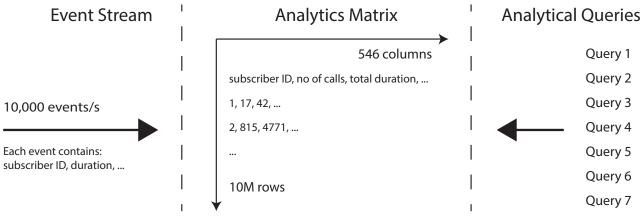


Figure 2: The AIM-Huawei workload

Analytics Matrix, is a materialized view on a large number of aggregates for each individual subscriber. There is an aggregate for each combination of aggregation function (min, max, sum), aggregation window (this day, this week, ...) and several event attributes as shown in Table 2, which shows a small part of the conceptual schema of an *Analytics Matrix*. For instance, there is an aggregate for the shortest duration of an international phone call today (attribute *min* in Table 2). The number of such aggregates (which defines the number of columns of the *Analytics Matrix*) is a workload parameter with default value 546, which we use in our experiments. The *Analytics Matrix* also contains foreign keys to dimension tables. Since these dimension tables are very small, we omit them in our experiments.

The use case requires two things to be done in real time: (a) update *Analytics Matrix* and (b) run analytical queries on the current state of the *Analytics Matrix*. (a) is referred to as Event Stream Processing (ESP) and (b) as Real-Time Analytics (RTA). When an event arrives in ESP, the corresponding record in the *Analytics Matrix* has to be atomically updated. RTA, on the other hand, is used to answer business intelligence questions. RTA queries are continuously being issued by one or multiple clients and are evaluated on a consistent state of the *Analytics Matrix*. This consistent state (or snapshot) is not allowed to be older than a certain bound t_{fresh}, which is a service level objective (SLO) of the Huawei-AIM benchmark and defaults to one second. Table 3 shows the seven queries from the original benchmark [2]. Additionally, users may issue ad-hoc queries. Since ad-hoc queries are not available upfront and can involve any number of attributes, it is impractical for a stream processing system to create specialized index structures.

Figure 2 summarizes the workload components. Events are ingested at a specific rate f_{ESP}, which will usually be 10,000 events per second in our experiments. Each event consists of a subscriber ID and call-dependent details, such as the call's duration, cost, and type (i.e., local or international). The *Analytics Matrix* is the aggregated state on the call records as described earlier and consists of 546 columns and 10 million rows, each representing the state of one subscriber. Depending on the event details, the corresponding subset of columns in the *Analytics Matrix* is updated for the particular subscriber. These updates are made available to analytical queries within t_{fresh}.

```
Query 1:
SELECT AVG (total_duration_this_week)
FROM AnalyticsMatrix
WHERE number_of_local_calls_this_week > α;
Query 2:
SELECT MAX (most_expensive_call_this_week)
FROM AnalyticsMatrix
WHERE total_number_of_calls_this_week > β;
Query 3:
SELECT (SUM (total_cost_this_week)) /
    (SUM (total_duration_this_week)) as cost ratio
FROM AnalyticsMatrix
GROUP BY number_of_calls_this_week
LIMIT 100;
Query 4:
SELECT city, AVG(number_of_local_calls_this_week),
    SUM(total_duration_of_local_calls_this_week)
FROM AnalyticsMatrix, RegionInfo
WHERE number_of_local_calls_this_week > γ
   AND total_duration_of_local_calls_this_week > δ
   AND AnalyticsMatrix.zip = RegionInfo.zip
GROUP BY city;
Query 5:
SELECT region,
   SUM (total_cost_of_local_calls_this_week) as local,
   SUM (total_cost_of_long_distance_calls_this_week)
    as long_distance
FROM AnalyticsMatrix a, SubscriptionType t,
   Category c, RegionInfo r
   WHERE t.type = t AND c.category = cat,
   AND a.subscription type = t.id AND a.category = c.id,
   AND a.zip = r.zip
GROUP BY region;
Query 6:
report the entity-ids of the records with the longest call this day and
this week for local and long distance calls for a specific country cty
Query 7:
SELECT (SUM (total_cost_this_week)) /
    (SUM (total_duration_this_week))
FROM AnalyticsMatrix
WHERE CellValueType = v;
```

Table 3: RTA queries 1 to 7, $\alpha \in$ [0,2], $\beta \in$ [2,5], $\gamma \in$ [2,10], $\delta \in$ [20,150], *t* $\in$ SubscriptionTypes, *cat* $\in$ Categories, *cty* $\in$ Countries, *v* $\in$ CellValueTypes

3.2 Implementations

We implemented the workload using at least one representative of each of the three categories: MMDBs, modern streaming systems, and hand-crafted systems. We chose Flink as a representative modern streaming system since it features a continuous processing model combined with a batch-based fault-tolerance mechanism allowing for low latency under high throughput conditions. MemSQL currently does not support stored procedures[14]. Without this feature, we were not able to implement the event processing part of the workload in an efficient way and therefore decided not to further evaluate MemSQL.

3.2.1 HyPer

Our workload implementation in HyPer was based on the work of [2]. ESP is performed using a stored procedure that updates aggregates stored in the *Analytics Matrix*, which is implemented as a regular database table. RTA query processing is implemented using SQL queries on that table.

When HyPer was first evaluated using the Huawei-AIM benchmark in [2], HyPer was configured to use a *copy-on-write*-based snapshotting technique that forked a child from the main OLTP process at a specific time interval. This enables RTA queries to be executed on a consistent snapshot of the *Analytics Matrix*. Since the table representing the *Analytics Matrix* can be as large as 50 GBs, forking a child of the OLTP process (essentially a copy of its page table) may take up to a hundred milliseconds. Additionally, our workload updates the records of randomly selected subscribers at a rate of 10,000 events/s, which may impact performance as the *copy-on-write* mechanism copies updated pages to main-

[14] http://docs.memsql.com/docs/mysql-features-unsupported-in-memsql

tain consistent snapshots for RTA queries. HyPer currently does not implement physical *MVCC*[15], which would lead to better results than a *copy-on-write*-based approach. The evaluated implementation interleaves the execution of multiple analytical queries thereby hiding memory latencies and single-threaded phases (e.g., result materialization). Writes, however, are never executed at the same time than analytical queries.

HyPer implements the PostgreSQL wire protocol allowing one to use any PostgreSQL client. In our experiments, we used PostgreSQL's C++ library (pqxx) to communicate between clients and HyPer (using TCP over UNIX domain sockets). Since HyPer currently does not implement batched transactions, HyPer's event processing throughput would be purely limited by network round trips between subsequent write requests, context switches on the server to receive incoming requests, and deserialization costs. To simulate batch processing, we decided to additionally generate the events within HyPer and only process these. In other words, instead of actually transferring the batch of events from the client to the server, we send a request to generate and process a specified number of events.

3.2.2 *Tell*

For Tell, we used the Huawei-AIM benchmark implementation from the Tell GitHub project[16]. We configured TellStore to use the *ColumnMap* layout with a total of 84 GB of memory, more than twice the memory that HyPer uses. With less memory, TellStore regularly ran out of memory, especially with multiple storage threads. To minimize NUMA effects, we configured Tell to run the storage layer (TellStore) on NUMA node 0 and the compute layer (RTA and ESP server-side threads) on node 1. RTA and ESP clients were also run on node 1. With this configuration, Tell achieved significantly better numbers than with the non-NUMA-aware configuration.

It is worth mentioning that Tell, as opposed to AIM, cannot be deployed in standalone mode. Whereas in AIM, Flink, and HyPer events are generated internally, Tell needs a client that generates events and sends them to the server (using UDP over Ethernet). Additionally, the server needs to send read and write requests to the storage (using RDMA over InfiniBand). Compared to all other implementations presented in this section, this makes ESP much more expensive as the overheads of network costs, context switching, and deserialization cost are paid twice (cf., Section 3.2.1). These extra costs should be be taken into account when looking at the performance results.

As Tell is a layered system, we have to carefully allocate threads to layers. In the compute layer, we have to allocate a specific number of ESP and RTA processing threads, whereas in the storage layer, we have to allocate the right number of scan threads (responsible for analytical query processing). The storage layer also runs one thread that integrates updates into the next snapshot for analytics and one thread for garbage collection. Microbenchmarks revealed the optimal thread allocation strategy for each workload as shown in Table 4. In general, Tell has a lot of different parameters most of which relate to memory management; and fine-tuning these parameters to get the best performance was a tedious task.

	Compute		**Storage**			
Workload	**ESP**	**RTA**	**scan**	**update**	**GC**	**Total**
read/write	1	n	n	1	1	$2n+2^*$
read-only	0	n	n	0	0	$2n$
write-only	n	0	0	1	0	$n+1$

Table 4: Thread allocation strategy for different workloads

3.2.3 *AIM*

Since the AIM system was specifically designed to address the AIM-Huawei workload, we assumed that it would achieve the best performance on the full workload and thus we used it as a baseline for our experiments. We used the same version used in [2] but in standalone mode where client and server communicate through shared memory. For the overall and the read-only experiments, we increased the number of RTA threads (and used one ESP thread), whereas for the write-only experiments, we increased the number of ESP threads.

3.2.4 *Flink*

Currently, Flink does not support exposing its internal state to external analytical queries. There is, however, a pull request for a partitioned key-value store that will be queryable. However, this queryable state only supports point lookups and thus cannot be used to implement the AIM workload. We implemented a custom operator that supports table scans to meet the requirements of the AIM workload. We experimented with a row and a column store layout for storing the state. Since the AIM workload is mostly analytical, we opted for the column store layout.

Similar to HyPer, we generated the events internally in Flink. We also implemented a version that uses Kafka for event ingestion, which will not be included in the results, as we found no significant difference in performance compared to the version that generates the events internally. In production, Kafka, or any other durable data source, is preferable to ensure full fault tolerance.

Since we want to make the most recent state available to analytical queries, windows need to be computed on an event basis. As Flink's built-in operators are not optimized for these continuous window computations, we chose to manually implement the window logic, which yielded better results. We did not enable Flink's checkpointing mechanism since the processing state of the Huawei-AIM workload can be as large as 50 GBs. Persisting a state of this size would lead to a significant performance penalty.

Flink provides many built-in functionalities that seem suitable for our workload including windowed streams supporting various aggregation functions (e.g., min, max, and sum). We tried to make use of the provided functionalities. However, in the studied version of Flink, combining multiple aggregation functions that produce only one single output stream is not yet supported. For this reason, we implemented a custom aggregation operator.

[15] [15] explains how versioned positions allow for fast scans.

[16] https://github.com/tellproject/aim-benchmark

*Since GC is only running from time to time and the update thread is also mostly idling for 10,000 events per second, both threads have an average CPU usage clearly below 50%. This is why we count them as one.

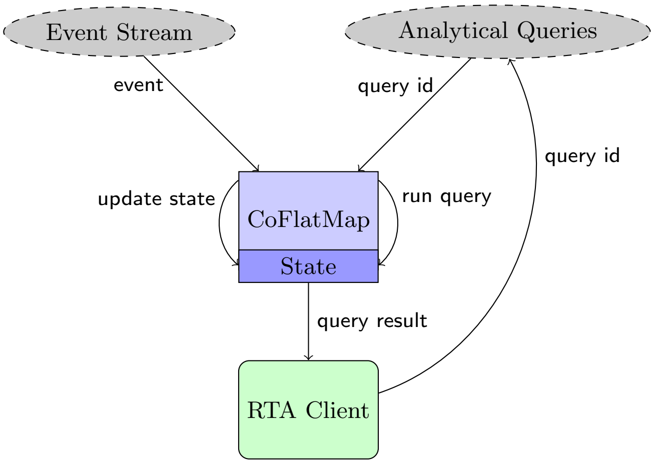


Figure 3: Hybrid processing in Flink. A CoFlatMap operator interleaves events with analytical queries.

All aggregations in the AIM workload are windowed. We could express this behavior using Flink's built-in window operators. For only one window type, this works well. However, with two or more different window types, the different windows would need to be merged into one consistent state across all windows. As this is not a straightforward operation in Flink, we decided to implement windows ourselves.

Another challenge was to run the analytical queries on the state maintained by the event processing pipeline. Flink does not provide a globally accessible state that can be used in such cases. States are only maintained at an operator level and cannot be accessed from outside. We solved this problem by processing both the event stream and the analytical queries in the same CoFlatMap operator as shown in Figure 3. Both streams are processed interleaved using two individual FlatMap functions that both work on the same shared state. This works as both functions are part of the same operator. Our implementation interleaves the two different streams on a partition basis. Since Flink follows the embarrassingly parallel paradigm, it is not designed to synchronize access across partitions. As described in [2], the AIM-Huawei workload does not require such a global synchronization since events are only ordered on an entity basis.

A powerful feature of Flink is its partitioning. Flink automatically partitions elements of a stream by their key and assigns the partitions to a parallel instance of each operator. Each instance of our CoFlatMap operator only receives the events for its partition and thereby maintains a part of the total state. The analytical queries, however, should run on the whole state. Therefore, we broadcast the queries to each CoFlatMap operator instance and run them on the individual partitions. The resulting partial results are merged in a subsequent operator.

In our experiments, we used Kafka to send queries since it integrates well with Flink and ensures that no queries are lost. It would also be possible to ingest the queries using a TCP client or other more sophisticated handwritten clients.

4. PERFORMANCE EVALUATION

We begin with a performance evaluation using the complete Huawei-AIM workload as described in Section 3. We drill down into the different aspects of the workload, including updates and real-time analytics. We then investigate the performance impact of the number of clients and the number of maintained aggregates.

System	Version
HyPer	Sep 12, 2016
Tell	0.2
AIM	Same version as used in [2]
Flink	1.1-Snapshot

Table 5: Evaluated systems

We did not evaluate how the systems scale out to multiple machines since the evaluated version of HyPer is standalone. In future work, we plan to extend HyPer with distributed event processing. However, the boundary between distributed systems and single-machine many-core systems with non-uniform memory access is blurry. In fact, as shown in the following section, even on our single two-socket machine the performance dropped when scaling beyond a single NUMA node.

4.1 Configuration

We evaluated the different systems (cf., Table 5) on an Ubuntu 15.10 machine with an Intel Xeon E5-2660 v2 CPU (2.20 GHz, 3.00 GHz maximum turbo boost) and 256 GB DDR3 RAM. The machine has two NUMA sockets with 10 physical cores (20 hyperthreads) each, resulting in a total of 20 physical cores (40 hyperthreads). The sockets communicate using a high-speed QPI interconnect (16 GB/s).

We placed the clients on the same machine as the server and generated events and queries by one client thread each (except for Tell where we used eight RTA client threads). Setting the total number of threads[17] was enough to run HyPer and Flink out-of-the-box. Conversely, Tell and AIM required more tedious fine-tuning and server threads were allocated as explained in Sections 3.2.2 and 3.2.3. As one can see from these allocation schemes, some workloads require more than one thread even in the most basic setting, which is why the measurements for AIM and Tell do not typically start at one thread and may have gaps.

4.2 Overall Performance

Figure 4 illustrates the query throughput when running the full workload, which consists of 10M subscribers, 10,000 events per second, and the seven analytical queries (cf., Table 3) where each of them is executed with equal probability. Further, daily and hourly windows are maintained leading to a total of 546 aggregates. AIM achieved the best performance. With two threads, it had a throughput of 14.8 queries/s and its best throughput, with eight threads, was 145 queries/s. The reason why AIM achieves its best performance at eight (and not at ten) threads is a NUMA effect: Since AIM statically pins threads to cores and allocates memory locally whenever possible, the total number of client and threads $(2 + 8 = 10)$ precisely fits on NUMA node 0. Hence, there is no communication to a remote memory region as it is the case for nine and ten threads. The spike at four threads probably relates to non-uniform communica-

[17] Unless otherwise noted, we are always referring to the server-side threads.

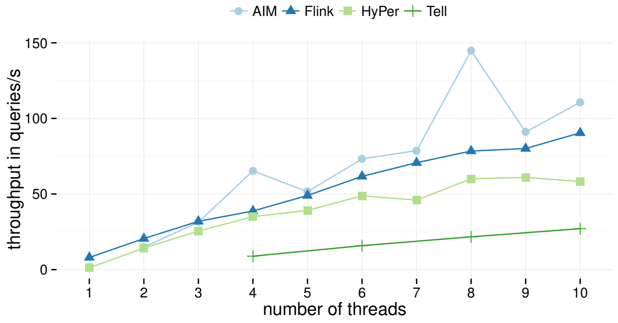


Figure 4: Analytical query throughput for 10M subscribers at 10,000 events/s

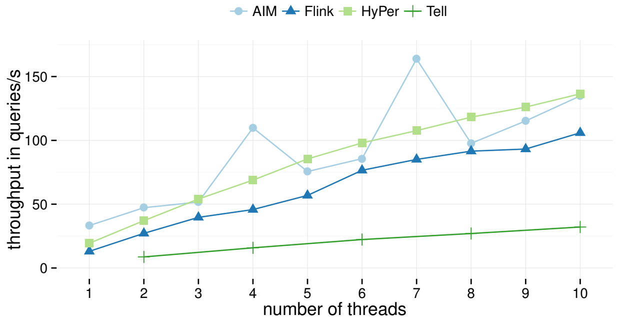


Figure 5: Analytical query throughput for 10M subscribers

tion paths between the cores on NUMA node 0. The spikes observed here are reproducable, and are, as we will see, also present in other workloads. Flink matches the performance of AIM for two threads and scales up to 90.5 queries/s using ten threads. HyPer achieved a throughput of 14.3 and 70.0 queries/s with two and nine threads, respectively. HyPer's throughput is lower than AIM's since it interleaves analytical queries with writes (i.e., writes block reads) while AIM processes them in parallel. With four threads, Tell achieved a query throughput of 8.90 queries/s and 27.1 queries/s with ten threads.

4.3 Read Performance

Figure 5 shows the analytical query throughput for the different systems with an increasing number of threads without concurrent events. With one thread, HyPer processed 19.4 queries/s while AIM sustained a throughput of 33.3 queries/s. As we increased the number of threads, HyPer sometimes outperformed AIM and its throughput increased linearly while AIM showed the same spikes as before[18]. HyPer's maximum throughput was 136 queries/s with ten threads compared to 164 queries/s for AIM with seven threads. Flink's throughput was 13.1 queries/s using one thread and gradually increased to 105.9 queries/s with ten threads. With two threads, Tell sustained a throughput of 8.68 queries/s while its maximum throughput was 32.1 queries/s using ten threads.

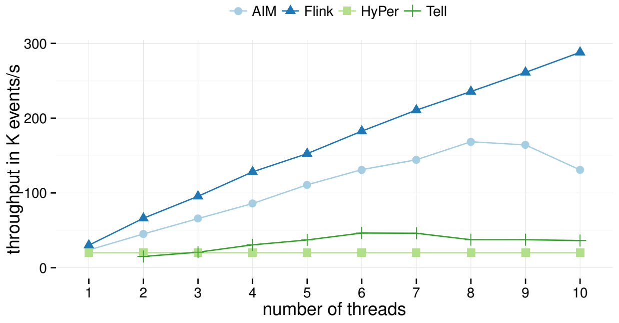


Figure 6: Event processing throughput with an increasing number of event processing threads

4.4 Write Performance

Figure 6 shows the event processing throughput of the different systems with an increasing number of event processing threads. This time, we evaluated the systems purely on the basis of their write throughput without running any analytical queries in parallel. Flink achieved the best write performance by far. Using one thread, it had a throughput of 30,100 events/s and the throughput scaled almost linearly to 288,000 events/s using ten threads. There are two reasons for this: (1) Flink partitions the state depending on the number of available processing threads. With this strategy, it scales well since there is no cross-partition synchronization involved. (2) Flink does not have any overhead introduced by snapshotting mechanisms or durability guarantees. AIM processed 23,700 events/s using one thread and achieved a maximum throughput of 168,000 events/s using eight threads, roughly 1.7x less than Flink. Again, we see the NUMA effect described earlier. AIM also partitions the state to scale its write throughput, but since its *differential update* mechanism introduces an overhead, AIM did not perform as well as Flink. Tell was able to process up to 46,600 events/s using six threads. The reason for the performance degradation after six threads is again a NUMA effect. All ESP processing threads as well as threads that handle UDP events are allocated on NUMA node 1 leading to an oversubscription of cores. HyPer sustained a throughput of 20,000 events/s in all cases since it only uses one single thread to process transactions.

4.5 Query Response Times

In this experiment, we measured the response time for each of the seven analytical queries with and without concurrent writes (10,000 events/s) using four threads. Table 6 shows the individual query response times and the overall average. HyPer's performance degraded the most when writes were added to the query processing workload. The reason is that HyPer interleaves analytical queries with writes. As shown in the previous section, HyPer's write throughput is limited to 20,000 events/s and does not scale for multiple threads. Thus, an event throughput of 10,000 events/s blocks the query processing for about 500 ms ev-

[18] AIM cannot be configured with zero ESP threads, which is why there is an additional idle ESP thread that we do not account for, but which nevertheless occupies its CPU. This is why the spike is at seven threads this time.

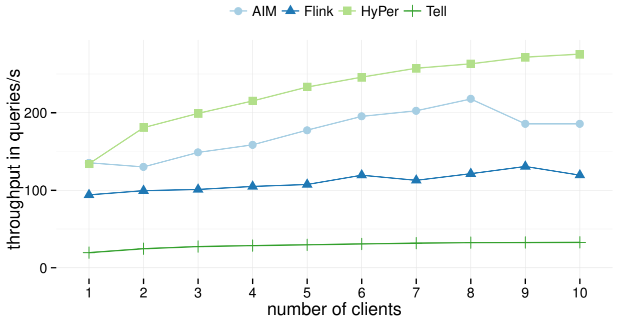


Figure 7: Analytical query throughput with an increasing number of clients

ery second. The query processing can only happen in the remaining 500 ms. AIM and Tell did not experience the same performance degradation since they perform writes and reads in parallel using the *differential updates* approach. Flink's performance does not drop much when adding 10,000 events/s as its (parallel) write throughput is so high that the analytical queries remain almost unaffected. However, we expect a higher performance degradation in Flink's analytical performance when increasing the number of events per second as Flink lacks efficient snapshotting mechanisms.

4.6 Impact of Number of Clients

Figure 7 shows the analytical query throughput with an increasing number of clients using ten server-side threads. HyPer performed the best of all systems and achieved a maximum throughput of 276 queries/s with ten client threads. HyPer's performance improves with multiple clients since it interleaves the execution of analytical queries (cf., Section 3.2.1). AIM's peak throughput was 218 queries/s with eight client threads. The gradual increase in the throughput shows the effect of the *shared scan* technique as AIM can now batch queries from multiple clients and process them all at once. The fact that the performance drops after eight threads shows that batching is only beneficial up to a certain point. Flink executes analytical queries as follows: Once a worker completed its part of the query, i.e., processed the query on its partition of the state, the worker can continue with the next query. The worker does not have to wait until the other partitions have been processed and the partial query results have been merged. For this reason, the idle time of threads decreases for more clients and the query throughput increases to 131 queries/s. Tell employs the same strategy as AIM and we can see a similar gradual increase in its throughput.

4.7 Impact of Number of Aggregates

In this experiment, we studied the impact of the number of aggregates being maintained. We measured the overall as well as the write performance of AIM, HyPer, and Flink while maintaining 42 instead of the original 546 aggregates. We did not measure Tell here since its AIM benchmark implementation was not flexible enough to accommodate schema changes.

Figure 8 shows the analytical query throughput for 10M subscribers and 42 aggregates at 10,000 events/s. Again,

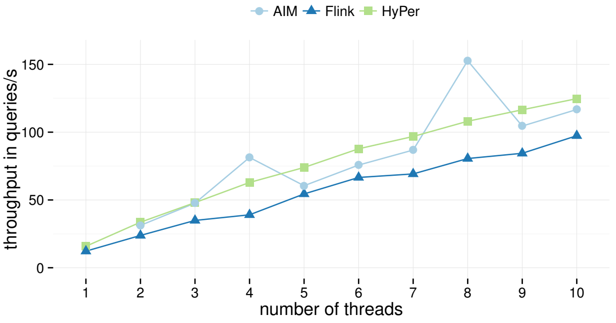


Figure 8: Analytical query throughput for 10M subscribers and 42 aggregates at 10,000 events/s

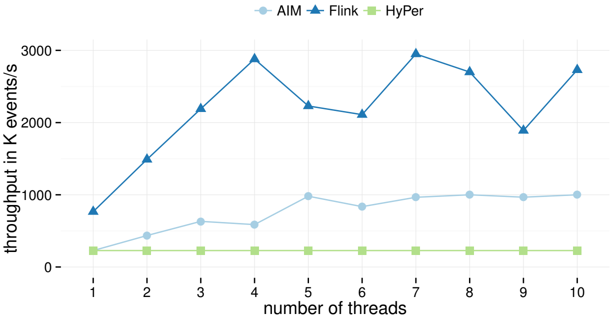


Figure 9: Event processing throughput for 42 aggregates with an increasing number of event processing threads

AIM achieved its best performance at eight threads (cf., Section 4.2) whereas Flink and HyPer did not experience such spikes. In contrast to the overall workload with 546 aggregates, HyPer achieved a higher performance than Flink throughout this experiment. The gain in HyPer's performance is expected since writes are now less expensive and thus singlethreaded phases are reduced. With ten threads, HyPer achieved a throughput of 125 queries/s (2.14x speedup over 546 aggregates) while Flink sustained 97.4 queries/s (1.08x).

Figure 9 shows the event processing throughput for 42 aggregates with an increasing number of event processing threads. Note that we reduced the number of aggregates by a factor of 13. As expected, the throughputs improved significantly with less aggregates (cf. Section 4.4). With one thread, AIM and HyPer achieved a throughput of 227,000 (11.4x) and 228,000 events/s (9.62x), respectively, whereas Flink sustained 766,000 events/s (25.5x). With ten threads, AIM and Flink reached a throughput of 1,000,000 (7.69x) and 2,730,000 events/s (9.51x), respectively. HyPer's performance did not increase with more threads since it currently does not parallelize transactions.

5. CLOSING THE GAP

We have shown that general-purpose MMDBs perform fairly well on streaming workloads. Nevertheless, our experiments indicate that there is still a gap between the per-

	Read (in isolation)				Overall (w/ concurrent events)			
Query	**HyPer**	**Tell**	**AIM**	**Flink**	**HyPer**	**Tell**	**AIM**	**Flink**
Query 1	5.25	249	2.44	5.83	12.2	242	5.32	16.9
Query 2	7.41	241	3.91	5.10	14.3	253	4.94	8.03
Query 3	20.4	298	10.4	29.9	29.5	289	10.5	37.2
Query 4	4.05	269	2.98	3.14	12.1	281	4.67	6.97
Query 5	12.5	264	21.1	37.8	20.7	271	38.3	45.1
Query 6	33.8	505	13.8	24.4	84.1	492	54.4	33.6
Query 7	17.7	246	9.04	24.4	25.8	236	17.5	32.8
Average	14.4	296	9.10	18.7	28.4	295	19.3	25.8

Table 6: Query response times in milliseconds

formance and usability of MMDBs and modern streaming systems, such as Flink, particularly when it comes to scalable event processing (cf., Section 4.4).

We propose a threefold approach to improve the overall write performance of MMDBs: (a) improve single-threaded write performance, (b) use all cores on a single machine, and (c) distribute load across multiple machines. In the following, we will describe each of these aspects in more detail.

Event ingestion in modern streaming systems is usually implemented using a durable data source, such as Kafka. This allows the streaming system to neglect durability, leading to higher throughputs. Durability in these systems is usually more coarse-grained than in MMDBs. To match Flink's single-threaded write performance, MMDBs would need to offer a more coarse-grained durability level by using durable data sources instead of employing fine-grained redo log mechanisms.

AIM, Flink, and Tell are capable of processing events in parallel, whereas HyPer processes transactions in a single thread. To match their scalability, HyPer would need to be extended with parallel single-row transactions, which are less complicated to parallelize than full transactions. Such a streaming-optimized transaction isolation would only ensure that there are no conflicts on the primary key column(s). Work on allowing concurrent write transactions in HyPer is ongoing [9]. In Tell, for instance, batches of events are processed within the scope of a transaction and several transactions can be executed in parallel. The latter, however, comes at the high price of maintaining multiple versions of the data, which again reduces performance as our experiments illustrate.

MMDBs also need to be able to distribute writes across multiple machines. HyPer, for instance, could employ a similar strategy as Flink, which partitions the event input stream and distributes it across nodes. Towards this end, HyPer could employ the ScyPer architecture as suggested in [13], where transactions are processed by the primary ScyPer node, which multicasts redo logs to secondary nodes. These secondaries are dedicated to query processing thus freeing resources and leading to higher throughput rates on the primary node. To scale out writes as well as reads, these two strategies could be combined by having multiple event processing nodes (the primary node in the ScyPer architecture), each of them being responsible for a subset of events. These event processing nodes would then multicast their redo logs to query processing nodes (secondary nodes in the ScyPer architecture). We plan to investigate such an architecture in future research.

From a usability perspective, modern streaming systems offer many features that help users to set up streaming applications, such as their out-of-the-box support for sliding and tumbling windows. On the one hand, MMDBs support arbitrary SQL allowing users to customize the analytical parts of their workloads and to issue ad-hoc queries. On the other hand, adding windowed aggregation functions using stored procedures is a cumbersome task. PipelineDB[19], which is built on top of PostgreSQL, solves this usability issue by extending SQL with streaming features but still cannot match the performance of dedicated streaming systems as we found in early experiments. In addition to out-of-the-box streaming features, modern streaming systems allow users to add custom code. There has been work to allow for the same in MMDBs, such as the integration of high-level programming languages (using user-defined functions). These additions, however, still do not allow for the same flexibility as writing plain old Java code. These limitations are mainly caused by the multi-tenant nature of database systems and the security level that these systems need to fulfill. MMDBs would need to allow for optionally disabling security arrangements (e.g., enforcing access rights) in favor of better extensibility. Another mitigation path that MMDBs could follow is to simply add more streaming features to its SQL processing logic, namely, window-based semantics as proposed by PipelineDB and StreamSQL [16]. This is also a topic we plan to address in future research.

Besides extending MMDBs to better support streaming use cases, the gap between MMDBs and streaming systems could be closed from the other direction, which would mean extending streaming systems with additional storage management features and query mechanisms. There is ongoing work to make use of Apache Calcite[20] (a SQL parser and optimizer framework) to extend Flink with streaming SQL and query optimization capabilities. Cache and register locality are crucial for high query performance and they can both be addressed very efficiently by compiling query plans into native code [14]. While this was possible to achieve in systems like HyPer or Tell, which are written in C++ and LLVM, it is more difficult to implement using JVM-based systems such as the streaming systems evaluated in this paper. Moreover, implementing efficient storage management capabilities is a tedious task in JVM-based languages because to ensure data locality, custom memory management has to be implemented outside the JVM heap.

[19] https://www.pipelinedb.com/

[20] https://calcite.apache.org

6. CONCLUSIONS

In this paper, we evaluated a broad set of architectures to address *analytics on fast data*. We performed an experimental evaluation including at least one representative of each architecture. Our experiences as well as the performance results indicate that there still exists a gap between MMDBs and dedicated streaming systems. MMDBs are built for multi-tenant environments where durability and isolation guarantees are essential. Dedicated streaming systems, on the other hand, often compromise these guarantees in exchange for better performance or higher flexibility. To catch up with these systems, MMDBs need to be able to optionally lower their guarantees. In addition, new architectures are required that allow MMDBs to scale both their event and query processing performance to keep up with the demands for increasingly high event throughput rates. The question remains whether these additions as well as new architectures can enable MMDBs to address a broad set of streaming workloads. Once fully implemented, we plan to evaluate HyPer with enabled *MVCC* to investigate the impact of event processing on analytics including the effects of the garbage collection overhead *MVCC* introduces.

7. ACKNOWLEDGEMENTS

We want to thank Gyula Fóra for his help in optimizing our workload implementation in Flink. This work has been sponsored by the German Federal Ministry of Education and Research (BMBF) grant FASTDATA 01IS12057. This work is further part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs and Media, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government.

8. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 169–180. ACM, 2001.

[2] L. Braun, T. Etter, G. Gasparis, M. Kaufmann, D. Kossmann, D. Widmer, A. Avitzur, A. Iliopoulos, E. Levy, and N. Liang. Analytics in motion: High performance event-processing and real-time analytics in the same database. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 251–264. ACM, 2015.

[3] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Data Engineering*, page 28, 2015.

[4] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.

[5] 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.

[6] 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):1259–1270, 2014.

[7] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, pages 195–206, Apr. 2011.

[8] 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*, 5(1):61–72, 2011.

[9] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*, pages 3:1–3:8. ACM, 2016.

[10] S. Loesing, M. Pilman, T. Etter, and D. Kossmann. On the design and scalability of distributed shared-data databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 663–676. ACM, 2015.

[11] F. Mattern and C. Floerkemeier. From the internet of computers to the internet of things. In *From active data management to event-based systems and more*, pages 242–259. Springer, 2010.

[12] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, et al. S-store: Streaming meets transaction processing. *Proceedings of the VLDB Endowment*, 8(13):2134–2145, 2015.

[13] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann. ScyPer: A Hybrid OLTP&OLAP Distributed Main Memory Database System for Scalable Real-Time Analytics. In *BTW*, 2013.

[14] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[15] T. Neumann, T. Mühlbauer, and A. Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*, pages 677–689, New York, NY, USA, 2015. ACM.

[16] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[17] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[18] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable Performance for Unpredictable Workloads. *PVLDB*, 2(1):706–717, 2009.

[19] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

Self-managed collections: Off-heap memory management for scalable query-dominated collections

Fabian Nagel
University of Edinburgh, UK
F.O.Nagel@sms.ed.ac.uk

Gavin Bierman
Oracle Labs, Cambridge, UK
Gavin.Bierman@oracle.com

Aleksandar Dragojevic
Microsoft Research,
Cambridge, UK
alekd@microsoft.com

Stratis D. Viglas
University of Edinburgh, UK
sviglas@inf.ed.ac.uk

ABSTRACT

Explosive growth in DRAM capacities and the emergence of language-integrated query enable a new class of managed applications that perform complex query processing on huge volumes of data stored as collections of objects in the memory space of the application. While more flexible in terms of schema design and application development, this approach typically experiences sub-par query execution performance when compared to specialized systems like DBMS. To address this issue, we propose self-managed collections, which utilize off-heap memory management and dynamic query compilation to improve the performance of querying managed data through language-integrated query. We evaluate self-managed collections using both microbenchmarks and enumeration-heavy queries from the TPC-H business intelligence benchmark. Our results show that self-managed collections outperform ordinary managed collections in both query processing and memory management by up to an order of magnitude and even outperform an optimized in-memory columnar database system for the vast majority of queries.

1. INTRODUCTION

This work follows two recent trends in data management and query processing: language-integrated query and ever-increasing memory capacities.

Language-integrated query is the smooth integration of programming and database languages. The impedance mismatch between these two classes of languages is well-known, but recent developments, notably Microsoft's LINQ and, to a lesser extent, parallel streams and lambdas in Java, enrich the host programming language with relational-like query operators that can be composed to construct complex queries. Of particular interest to this work is that these queries can be targeted at both in-memory and external database data sources.

Over the last two decades, DRAM prices have been dropping at an annual rate of 33%. As of September 2016, servers with a DRAM capacity of more than 1TB are available for under US$50k. These servers allow the entire working set of many applications to fit into main memory, which greatly facilitates query processing as data no longer has to be continuously fetched from disk (*e.g.,* via a disk-based external data management system); instead, it can be loaded into main memory and processed there, thus improving query processing performance.

Granted, the use-case of a persistent (database) and a volatile (application) representation of data, coupled with a thin layer to translate between the two is how programmers have been implementing applications for decades and will certainly not go away for all existing legacy applications that are in production. Combining, however, the trends of large memories and language-integrated query is forward-looking and promises a novel class of *new* applications that store huge volumes of data in the memory space of the application and use language-integrated query to process the data, *without* having to deal with the duality of data representations. This promises to facilitate application design and development because there is no longer a need to setup an external system and to deal with the interoperability between the object-oriented application and the relational database system. Consider, for example, a business intelligence application that, on startup, loads a company's most recent business data into collections of managed objects and then analyses the data using language-integrated query. Such applications process queries that usually scan most of the application data and condense it into a few summarizing values that are then returned to the user; typically presented as interactive GUI elements such as graphs, diagrams or tables. These queries are inherently very expensive as they perform complex aggregation, join and sort operations, and thus dominate most other application costs. Therefore, fast query processing for language-integrated query is imperative.

Unfortunately, previous work [12, 13] has already shown that the underlying query evaluation model used in many language-integrated query implementations, *e.g.,* C$^\sharp$'s LINQ-to-objects, suffers from various significant inefficiencies that hamper performance. The most significant of these is the cost of calling virtual functions to propagate intermediate

Series ISSN: 2367-2005

10.5441/002/edbt.2017.07

result objects between query operators and to evaluate predicate and selector functions in each operator. Query compilation has been shown to address these issues by dynamically generating highly optimized query code that is compiled and executed to evaluate the query. Previous work [13] also observed that the cost of performing garbage collections and the memory layout of the collection data which is imposed by garbage collection further restricts query performance. This issue needs to be addressed to make this new class of applications feasible for application developers.

Our solution to address these inefficiencies is to use *self-managed collections* (SMCs), a new collection type that manages the memory space of its objects in private memory that is excluded from garbage collection. SMCs exhibit different collection semantics than regular managed collections. This semantics is derived from the table type in databases and allows SMCs to automatically manage the memory layout of contained objects using the underlying type-safe manual memory management system. SMCs are optimized to provide fast query processing performance for enumeration-heavy queries. As the collection manages the memory layout of all contained objects and is aware of the order in which they are accessed by queries, it can place them accordingly to better exploit spatial locality. Doing so improves the performance of enumeration-heavy queries as CPU and compiler prefetching is better utilized. This is not possible when using automatic garbage collection as the garbage collector is not aware of collections and their content. Objects may be scattered all over the managed heap and the order they are accessed may not reflect the order in which they are stored in memory. SMCs are designed with query compilation in mind and allow the generated code low-level access to contained objects, thus enabling the generation of more efficient query code. On top of this, SMCs reduce the total garbage collection overhead by excluding all contained objects from garbage collection. With applications storing huge volumes of data in SMCs, this further improves application performance and scalability.

The remainder of this paper is organized as follows. In §2, we provide an overview of SMCs and their semantics before presenting a type-safe manual memory management system in §3. In §4, we introduce SMCs and show how they utilize our manual memory manager to improve query processing performance compared to regular collections that contain managed objects. Finally, we evaluate SMCs in §7 using microbenchmarks as well as some queries from the TPC-H benchmark. We conclude this work in §9.

2. OVERVIEW

SMCs are a specialized collection type designed to provide improved query processing performance compared to regular managed collections for application data accessed predominantly by language-integrated queries. This performance improvement may come at the expense of the performance of other access patterns (*e.g.,* random access). SMCs are only meant to to be used with data that is dominantly accessed in queries.

SMCs have a new semantics: they own their contained objects and hence the collection itself determines the lifetime of the objects. In other words, objects are created when they are inserted into the collection and their lifetime ends with their removal from the collection. This accurately models many use cases, as objects often are not relevant to the application once they are removed from their host collection. Consider, for example, a collection that stores products sold by a company. Removing a product from the collection usually means that the product is no longer relevant to any other part of the application. Managed applications, on the other hand, keep objects alive so long as they are still referenced. This means that a rogue reference to an object that will never be touched again prevents the runtime from reclaiming the object's memory. Object containment is inspired by database tables, where removing a record from a table entirely removes the record from the database.

The following code excerpt illustrates how the `Add` and `Remove` methods of SMCs are used:

```
Collection<Person> persons = new Collection<Person>();
Person adam = persons.Add("Adam", 27);
/* ... */
persons.Remove(adam);
```

The collection's `Add` method allocates memory for the object, calls the object's constructor, adds the object to the collection and returns a reference to the object. As the lifetime of each object in the collection is defined by its containment in the collection, mapping the collection's `Add` and `Remove` methods to the `alloc` and `free` methods of the underlying memory manager is straightforward. When the `adam` object is removed from the collection, it is gone; but it may still be referenced by other objects. Our semantics requires that all references to a self-managed object implicitly become `null` after removing the object from its host collection; dereferencing them will throw a `NullReferenceException`.[1]

SMCs are intended for high-performance query processing of objects stored in main memory. To achieve this, they leverage query compilation [12, 13] and support bag semantics which allows the generated queries to enumerate a collection's objects in memory order. In order to exclude SMCs from garbage collection we have to disallow collection objects to reference managed objects. We enforce this by introducing the `tabular` class modifier to indicate classes backed by SMCs and statically ensure that tabular classes only reference other tabular classes. Strings referenced by tabular classes are considered part of the object; their lifetime matches that of the object, thereby allowing the collection to reclaim the memory for the string when reclaiming the object's memory. We further restrict SMCs not to be defined on base classes or interfaces, to ensure that all objects in a collection have the same size and memory layout.

In contrast to regular managed collection types like `List<T>` our collection types require a deeper integration with the managed runtime. As collections allocate and free memory for the objects they host, we introduce an off-heap memory system to the runtime that provides type, memory and thread safety. The `alloc` and `free` methods of the memory system are part of the runtime API and are called by the collection implementation as needed. The type safety guarantees for tabular types are not the same as for automatically managed ones. We guarantee that a reference always refers to an instance of the same type and that this instance is either the one that was assigned to the reference or, if the instance has been removed from the collection, `null`. This differs from automatically managed types that

[1] This suggests that an ownership type system could be useful to statically guarantee such exceptions are not raised; but we leave this to future work.

guarantee that a reference points to the object it was assigned to for as long as the reference exists and refers to that object. To ensure type-safe reference accesses, we store additional information with each reference and perform extra checks when accessing an object. For managed types, references are translated into pointer-to-memory addresses by the just-in-time (JIT) compiler. As the logic for tabular types is more complex, we modify the JIT compiler to make it aware of tabular type references and the code that must be produced when dereferencing them.

We use query compilation to transform LINQ queries on SMCs into query functions that process the query. To improve query performance, the generated code directly operates on the collection's memory blocks (using `unsafe`, C-style pointers). All objects in the collections are stored in memory blocks that are private to the collections. Note that these blocks are not accessible outside the collection and the code generator. We assume that the structure of most LINQ queries is statically defined in the application's source code with only query parameters (*e.g.,* a constant in a selection predicate) dynamically assigned. We modify the C$^\sharp$ compiler to automatically expand all LINQ queries on SMCs to calls to automatically generated imperative functions that contain the same parameters as arguments. Queries that are dynamically constructed at run-time, can be dealt with using a LINQ query provider as in [13]. The generated imperative query code processes the query as in [13], but on top of SMCs that enable direct pointer access to the underlying data.

3. TYPE-SAFE MANUAL MEMORY MANAGEMENT

Our manual memory management system is purpose-built for SMCs. It leverages various techniques to allow SMCs to manually manage contained objects and to provide fast query processing.

3.1 Type stability and incarnations

The memory manager allocates objects from unmanaged memory blocks, where each block only serves objects of a certain type. By only storing objects of a certain type in each block and disallowing variable-sized objects to be stored in-place we ensure that all object headers in a block remain at constant positions within that block, even after freeing objects and reusing their memory for new ones. We align the base address of all blocks to the block size to allow extracting the address of the block's header from the object pointer. This allows us to store type-specific information like `vtable` pointers only once per block rather than with every object. We refer to the memory space in a block that is occupied by an object as the object's *memory slot*. Object headers contain a 32-bit *incarnation number*. We use incarnations to ensure that objects are not accessed after having been freed. For each slot, the incarnation number is initialized to zero and incremented whenever an object is freed. References to objects store the incarnation of the object together with its pointer. Before accessing the object's data, the system verifies that the incarnation number of the reference matches that in the object's header and only then allows access to the object [1]. If the application tries to access an object that has been freed (*i.e.,* non matching incarnation numbers), then the system raises a null reference exception. The JIT compiler injects these checks when dereferencing a manually managed object. We do not expect incarnation numbers to overflow in the lifetime of a typical application, but if overflows should occur, we stop reusing these memory slots until a background thread has scanned all manually managed objects and has set all invalid references to `null`. Single-type memory blocks combined with incarnation numbers ensure type-safe manual memory management as defined in §2.

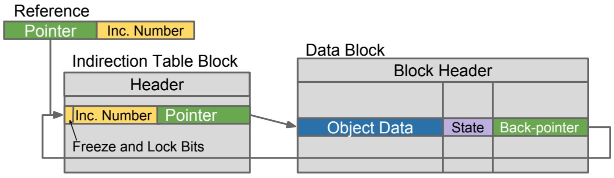


Figure 1: Accessing object data through indirection

3.2 Memory layout

We illustrate the memory layout of our approach in Figure 1. We do not store a pointer to an object's memory slot in its reference, but instead use a level of indirection. We will require this for the compaction schemes of §5. The pointer stored in object references points to an entry in the global indirection table which, in turn, contains a pointer to the object's memory slot. We store the incarnation number associated with an object in its indirection table entry rather than its memory slot. This allows us to reuse empty indirection table entries and memory blocks for different types without breaking our type guarantees.

As shown in Figure 1, each data block is divided into four consecutive memory segments: block header, object store, slot directory, and back-pointers. The object store contains all object data. Each object's data is accessible through a pointer from the corresponding indirection table entry or through the identifier of the object's slot in the block. The slot directory stores the state of each slot and further state-related information (for a total of 32 bits). Each slot can be in one of three states: *free* *i.e.,* the slot has never been used before, *valid*, *i.e.,* it contains object data, or *limbo* *i.e.,* the object has been removed, but its slot has not been reclaimed yet. Back-pointers are required for query processing and for compaction; they store a pointer to the object's indirection table entry. The slot directory entry and the back-pointer are accessible using the object's slot identifier.

3.3 Memory contexts

We have so far grouped objects of the same type in blocks private to that type. In many use cases, certain object types exhibit spatial locality: objects of the same collection are more likely to be accessed in close proximity. *Memory contexts* allow the programmer to instruct the allocation function to allocate objects in the blocks of a certain context (*e.g.,* a collection). The memory blocks of a context only contain objects of a single type and only the ones that have been allocated in that specific memory context.

3.4 Concurrency

Incarnation numbers protect references from accessing objects that have been freed. However, they do not protect objects from being freed and reused while being accessed. Consider Figure 2: Thread 2 frees and reuses the memory

Thread 1	Thread 2
`if (CHECK_INC(adam))`	
	`persons.Remove(adam);`
	`Person tom = persons.Add(‘‘Tom’’, 25);`
`PRINT(adam.name);`	

Figure 2: Concurrency conflict

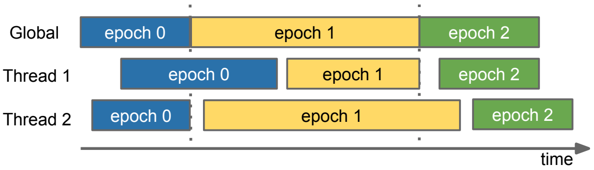


Figure 3: Epoch-based memory reclamation

slot referenced by the `adam` reference just after `Thread 1` successfully checked the incarnation numbers for the same object. As `Thread 1`'s incarnation number check was successful, the thread accesses the object, which is now no longer `Adam`, but `Tom`. This behavior violates the type-safety requirement of always returning the object assigned to a reference, or `null` if the referenced object has been freed. We refine the requirement for the concurrent case by specifying the check of the incarnation numbers to be the point in time where the requirement must hold. Thus, all accesses to objects are valid as long as the incarnation numbers matched at the time they were checked. To enforce the type-safety requirement, the memory manager ensures that if an object is freed, its memory slot cannot be reused for a new object until all concurrent threads have finished accessing that object.

We use a variation of epoch-based reclamation [7] to ensure thread safety. In epoch-based reclamation, threads access shared objects in grace periods (critical sections). The memory space of shared objects can only be reclaimed once all threads that may have accessed the object in a grace period have completed this grace period. Thus, grace periods are the time interval during which a thread can access objects without re-checking their incarnation numbers to ensure type safety. Epochs are time intervals during which all threads pass at least one grace period. The system maintains a global epoch; each thread maintains its thread-local epoch. In Figure 3, we show how we track epochs. Upon entering a critical section (grace period), each thread sets its thread-local epoch to the current global epoch. To leave a critical section, a thread can increment the global epoch if all other threads that currently are in critical sections have reached the current global epoch. Hence, threads can either be in the global epoch e or in $e - 1$. Memory freed in some global epoch e can safely be reclaimed in epoch $e+2$ because by that time, no concurrent thread can still be in epoch e.

To implement epoch-based reclamation, the JIT compiler automatically injects code to start and end critical sections when dereferencing manually managed objects. Critical sections are not limited to a single reference access; several accesses can be combined into a single critical section to amortize the overhead of starting and ending critical sections. The following illustrates the code to start and end a critical section:

```
void enter_critical_section() {
  global->sectionCtx[threadId].epoch = global->epoch;
  global->sectionCtx[threadId].inCritical = 1;
  memory_fence(); }

void exit_critical_section() {
  memory_fence();
  global->sectionCtx[threadId].inCritical = 0; }
```

Upon entering a critical section, each thread sets its local epoch to the current global epoch and sets a flag to indicate that the thread is currently in a critical section; on exit the thread clears this flag. We have to enforce compiler and CPU instruction ordering around these instructions to ensure that the session context is set before we access the object and not unset until we have finished, hence, the memory fences. In contrast to [7], we do not increment global epochs *modulo* three, but as a continuous counter. We also do not increment the global epoch and reclaim memory when exiting critical sections, but in the memory manager's allocation function. This allows us to lazily reclaim memory on demand when allocating new objects.

3.5 Memory operations

When freeing an object, we increment its incarnation number to prevent subsequent accesses to it. We refer to memory slots that are freed, but not yet available for reuse as *limbo* slots. We set the memory slot's state to limbo and set its removal timestamp to the current global epoch in the slot directory. This bookkeeping ensures that the slot cannot be reclaimed until at least two epochs have passed. Memory blocks become candidates for reclamation when they surpass a threshold fraction of limbo slots, the reclamation threshold. If this is the case, we add the block to a queue of same-type memory blocks that may be reclaimed, along with the earliest timestamp when the block can be reclaimed (global epoch plus two).

All allocations are performed from thread-local blocks so that only one thread allocates slots in a block at a time (though there can be concurrent removals from the same block). Thread-local blocks are taken from the reclamation queue of the appropriate type if there are blocks ready for reclamation; if the queue is empty they are allocated from the unmanaged heap. To find a memory slot for a new object the allocation function scans all entries in the slot directory from the slot of the last allocation until either a free slot or a reclaimable limbo slot is found. The maximum number of slots scanned before finding a limbo slot that can be reclaimed depends on the reclamation threshold. For instance, if blocks can host one hundred objects and are added to the queue once they contain more than 5% limbo slots, then each allocation scans at most twenty slots to find a reclaimable limbo slot. The actual number is likely to be smaller as removals might have happened in the meantime. The allocation function attempts to increment the global epoch counter once there are blocks in the reclamation queue that cannot be reclaimed yet because two epochs have not passed.

4. SELF-MANAGED COLLECTIONS

SMCs use the type-safe memory management described in §3 and support the semantics of §2. The objects contained in an SMC are managed by the collection itself and not by the garbage collector. This, along with bag semantics, enables SMCs to place objects in memory based on the order the objects are touched when enumerating the collection's content in a query. This improves the locality of memory accesses when enumerating the SMC, leading to improved performance compared to iterating over the collection's content through references that may point anywhere in the managed heap (as is the case for all conventional .NET collections). A convenient side-effect of disallowing SMCs to contain standard objects is that it significantly reduces the size of the managed heap and the volume of memory that has to be scanned during garbage collection and, in consequence, the duration of garbage collection, which improves the overall performance of the application.

SMCs use the type-safe memory manager of §3 to manage contained objects. The semantics of SMCs mean that the `Add` and `Remove` methods can directly be mapped to the memory manager's `alloc` and `free` methods. In addition to allocating memory for the object, the `Add` method calls the object's constructor and returns a reference to the object.

Each SMC has a private memory context to allocate all objects added to the collection. This ensures that all objects in an SMC end up in the same set of private memory blocks. The SMC can access all of these blocks through the memory context. Recall from §2 that we automatically transform LINQ queries over SMCs into calls to specialized query functions that use query compilation to improve the performance of query processing. By giving the SMC access to these memory blocks, we also allow the query compiler to access them to enumerate over the SMC's objects. The following illustrates a simple compiled query that enumerates over all objects in the SMC by iterating over all valid slots in all blocks in the SMC's memory context, checking a predicate on the `age` field, and returning references to all qualifying objects:

```
enter_critical_section();
foreach (Block* blk in collection.GetMemoryContext())
  foreach (Slot i in blk)
    if (blk->slots[i] == VALID)
      if (blk->data[i].age > 17)
        yield new ObjRef { ptr = blk->backptr[i],
                           inc = blk->backptr[i]->inc };
exit_critical_section();
```

The query uses the memory block's slot directory `blk->slots` to check if the corresponding memory slot contains a valid object (in contrast to a free or limbo slot). As each entry in the slot directory is only four bytes wide and stored in a consecutive memory area, it is fairly cheap to iterate over the slot directory to check for valid slots. The query touches the object's data only if the slot is valid. If the slot also satisfies the selection predicate, the query returns a reference (`ObjRef`) to the object. To do so, it uses the back-pointer field `blk->backptr` to obtain a pointer to the corresponding indirection table entry. The reference contains this pointer and the current incarnation number of the object to ensure that the memory slot can safely be reclaimed once the object is removed from the SMC. To generate code for more complex queries we follow a similar strategy as in previous work [10, 12, 13, 14].

To ensure that the accessed objects are not removed and their memory slot is not reclaimed while directly accessing objects in a query, we have to be in a critical section. This applies to objects in the primary SMC that we enumerate as well as to objects in other SMCs that we access through references from the primary SMC. Instead of entering and exiting a critical section around each object access, we process huge chunks of data in the same critical section. This amortizes the cost of critical sections (in particular, memory fences) and, hence, is a cornerstone of providing good query performance. The query remains in the same critical section either for its entire duration, or for the duration of processing a single memory block. The query compiler chooses the desired granularity for each query based on the requirements of the query. Staying in the same critical section for the duration of the query allows to generate code that stores direct pointers to the memory locations of SMC objects in intermediate results and data structures (otherwise the query may only use object references). However, it also increases the time until the memory manager can increment the global epoch to reclaim limbo slots. As LINQ queries are lazily evaluated, we enforce that critical sections are exited before a result object is returned and, hence, control is returned to the application. Since queries often contain several blocking operations (*e.g.,* aggregation or sorting), most query processing is performed in a single critical section. Objects that are concurrently removed from an SMC while a query enumerates the SMC's content are included in the query's result if: (*a*) the query reads the object's slot directory entry before the slot is set to limbo, or (*b*) the query follows a reference to the object before its incarnation number is incremented. Objects added to an SMC behave accordingly. SMCs use a lower isolation level than database systems, in line with other managed collections.

4.1 Columnar storage

While SMCs manage the memory space of contained objects themselves, they keep the memory layout of the object's data unchanged. Previous work in database systems, *e.g.,* [2], has shown that some workloads, however, greatly benefit from a columnar layout, instead of the row-wise layout of SMCs. Since SMCs store all object data in blocks that only contain objects from the same collection and, hence, the same type, they can be easily extended to leverage a columnar layout. The only requirements are that: (*a*) the JIT compiler injects the code required to access columnarly stored data when following references to such objects, and (*b*) the query compiler is aware of the data layout and also generates code that accesses the data in a columnar fashion. For columnar layouts, we store the object's block and slot identifiers in the object's indirection table entry instead of a pointer to the object's memory location. To access the data of an object, we look up its memory block using an array of memory blocks indexed by their block identifier, and then use the slot identifier to find the position of the value in its column.

5. COMPACTION

Common uses of SMCs do not cause them to shrink significantly; they stay at a stable size or grow steadily. However, when facing heavy shrinkage of an SMC, we perform compaction to reduce the SMC's memory footprint and improve query performance. When relocating objects as part of a compaction, we have to ensure that concurrent accesses to them do not exhibit inconsistencies. Inconsistencies may arise from accesses through references or from queries directly operating on the SMC's memory blocks.

5.1 Reference access

The indirection table allows us to move data objects within and across memory blocks without having to update all ref-

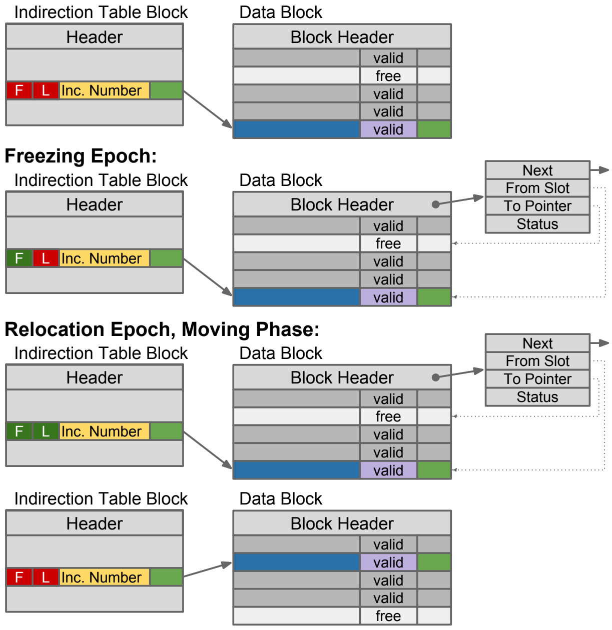


Figure 4: Relocating an object

erences held by the application. Atomically updating the pointer in the indirection table suffices to ensure that all threads can correctly reach the object. However, threads that already are in critical sections and point to the old location might cause inconsistencies by performing updates on outdated memory locations. To compact data blocks without stopping the application we extend the epoch scheme for object relocation. We reserve the two most significant bits of the incarnation number in the indirection table for a *frozen* flag [3] and a *lock* flag. After a thread successfully increments the global epoch, it checks if a compaction is necessary. The global epoch cannot be increased in the meantime because the thread is still in a critical section using the previous epoch. If compaction is necessary we set the global `nextRelocationEpoch` to $e + 2$ (e is the thread-local epoch and $e+1$ is the global epoch we just incremented) and then awake the compaction thread. Once a relocation epoch is set, no other but the compaction thread can increment the global epoch until the compaction is finished (epoch $e + 3$). To guarantee this, we run the compaction thread in a critical section that uses the thread-local epoch e, which prevents all other threads from incrementing the global epoch.

The compaction thread is active through two epochs: the freezing epoch $e + 1$ and the relocation epoch $e + 2$. In the freezing epoch it iterates over all blocks that need compaction (marked by previous allocations/removals). For each block, it constructs a list of all slots that have to be moved and the memory address the slots have to be moved to. This list is accessible through the block's header. The thread then sets the frozen bit in the indirection table entry of each slot that is scheduled to be copied.[2] Once all blocks are prepared for compaction, the thread waits until all other threads are in the freezing epoch ($e + 1$) and then increments the global epoch to $e + 2$ to start the relocation epoch. The relocation epoch consists of two phases: the waiting phase, which lasts until the compaction thread observes that all other threads are in the relocation epoch, and the moving phase that starts thereafter. While waiting, the compaction threat continuously tries to increment the global epoch to proceed to the moving phase. Once in the moving phase the compaction thread makes this phase globally visible by setting a global variable to indicate that frozen objects may now be moved. It then iterates over all blocks scheduled for compaction. For every slot to be moved, it atomically locks the incarnation number by setting the lock bit and copies the object to the new location, updates the pointer in the indirection table, unsets the lock and freeze bits, and marks the relocation as successful in the block's relocation list. Once all scheduled relocations are done, the compaction thread increments the global epoch to $e + 3$ (all threads are guaranteed to be at $e + 2$ by this point), exits its critical section to allow other threads to increment the global epoch, and goes back to sleep. Figure 4 illustrates the steps to move an object inside a memory block.

If an object's incarnation number is not frozen there is no risk of it being moved in the current epoch, so all threads can access it as before. Note that the incarnation number comparison that we have to do anyway is enough to cover the most common path. If we encounter a frozen incarnation number (*i.e.,* the first incarnation number comparison fails, but a second that excludes frozen and lock bits succeeds), there are three cases: (a) We are in the freezing epoch. There will not be any relocation in this epoch, so we can return the data pointer. (b) We are in the waiting phase of the relocation epoch and not all threads are in the relocation epoch yet. A relocation might happen while we access the object so we cannot proceed. However, we also cannot relocate the object because not all threads are in the relocation phase so they do not expect relocations yet. Our only option is to bail out from relocating the object. To do so, we find the object's entry in the block's relocation list, atomically set the lock bit in the object's incarnation number, set the status of the relocation to failed (in the object's relocation list entry), and unset the freeze and lock bits. If the lock bit has already been set by another thread, we spin until it is unset and then recheck the object's status. Once the freeze bit is removed, we can return the pointer and proceed. (c) We are in the moving phase of the relocation epoch and all other threads are also in the relocation epoch. We again cannot proceed because the object may be moved at any time, but we can help the compaction thread move the object to its new location and then proceed. To do so, we find the object's entry in the block's relocation list, atomically set the lock bit in the object's incarnation number, move the object to its new location, set the status of the relocation to succeeded, and unset the freeze and lock bits. As in the previous case, we spin if the bit is locked, then recheck its status and finally return the pointer after the frozen bit is unset. The following outlines the checks that have to be performed before accessing a manually managed objects through its reference:

[2] By using a CAS operation; this requires `free` to also use CAS to increment incarnation numbers

```
void* dereference_object(ObjRef oref) {
  if(oref.inc == oref.ptr->inc) {
    return oref.ptr->memptr;
  } else if (oref.inc == (oref.ptr->inc & FL_MASK)) {
    // First case:
    if (global->sectionCtx[threadID].epoch
         != global->nextRelocationEpoch) {
      return oref.ptr->memptr;
    // Second case:
    } else if (!global->inMovingPhase) {
      bail_out_relocation(oref);
      return oref.ptr->memptr;
    // Third case:
    } else {
      relocate_object(oref);
      return oref.ptr->memptr; }
  } else {
    throw new NullPointerException(); } }
```

Note that outside freeze and relocation epochs, the first condition is always satisfied if the referenced object has not been freed. If the object access is known to be read-only, we can always use the original location of the object in the waiting phase of the relocation epoch as its memory location cannot be reclaimed while we access it. In this case, the reader does not have to fail the relocation of that object.

When the compaction thread starts iterating over the blocks to be compacted (*i.e.,* the moving phase of the relocation epoch), all failed relocations are visible so the thread can deal with them. If necessary, it extends compaction by one additional epoch to try all unsuccessful relocations again by adding another freezing phase at the end of the relocation epoch and setting the following epoch to be a relocation epoch before exiting the current relocation epoch.

5.2 Block access

Queries directly operating on the memory blocks of an SMC can also cause inconsistencies where the query misses some objects because they are concurrently being relocated or includes them twice. To prevent these inconsistencies, we have to extend the compaction scheme described thus far. We always empty the memory blocks that take part in the compaction by moving their objects to new memory blocks and removing the emptied blocks from the collection. Blocks only participate in a compaction if their occupancy is below a threshold (*e.g.,* 30%). Blocks that participate in a compaction are assigned to *compaction groups* where the objects of all blocks in a compaction group are moved to the same new block. The number of blocks in a compaction group depends on the aforementioned threshold; a 30% threshold results in three blocks per group.

Queries process all blocks of a compaction group in the same thread-local epoch and in consecutive order. This ensures consistent query behavior outside relocation epochs as relocations may not start while processing the compaction group. During relocation epochs, we have to ensure that queries may either only access the pre-relocation state of a compaction group or the post-relocation state. If processing of a compaction group starts in the moving phase of the relocation epoch, the query first helps performing the relocation of the compaction group and then uses the compacted memory block for query processing. If processing of the group starts in the waiting phase, we cannot compact the group's content yet. In this case, we add the group to a list of groups that still have to be processed and continue with the remaining memory blocks. Once all remaining blocks are exhausted, we check if the moving phase has already started and, if this is the case, process all remaining compaction groups by first performing the relocation and then processing the compacted block. If the moving phase has not started yet, we process the compaction group in its pre-relocation state by atomically incrementing a query counter in the compaction group that prevents other threads from compacting the group until the query decremented the counter again. Relocations only occur in the moving phase of the relocation epoch and, hence, once a relocation waits for the query counter of a compaction group to become zero, there are no more queries incrementing it. The compaction thread bails out of compacting a certain group after waiting for a predefined amount of time for the read lock to be released. We do this to deal with queries that return control to the application (*i.e.,* return a result element) while holding the read lock.

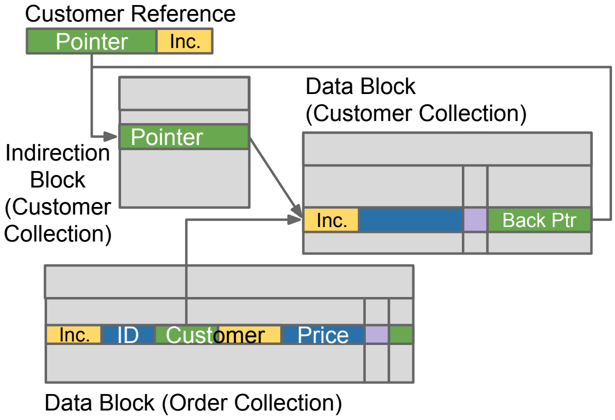

Customer Reference
Pointer
Inc.
Indirection Block (Customer Collection)
Pointer
Data Block (Customer Collection)
Inc.
Back Ptr
Inc.
ID
Customer
Price
Data Block (Order Collection)

Figure 5: Direct pointer between collection objects

6. DIRECT POINTERS

When a query touches an object that contains many references to nested objects, then SMCs may loose ground to automatically managed collections: each dereference not only has to check incarnation numbers, but, more importantly, it has to pay for an additional (random) memory access to the indirection table. We now provide an alternative implementation that solves this problem. We keep indirection for all external references, but, for references between SMCs, we store the direct pointer to the corresponding memory location. To be able to check incarnation numbers in both cases, the incarnation number of a memory slot is moved back into the memory slot (object header) instead of the indirection table. In Figure 5 we show the new layout, which improves query performance for queries that use references to access objects from several SMCs.

When relocating an object, however, the new memory location of the object now has to be updated in the indirection table as well as in all self-managed objects that reference it, which is no longer an atomic operation. We address this by adding a third flag to the incarnation number, the *forwarding* flag. The forwarding flag turns the object's old memory slot into a tombstone. Queries reaching the tombstone through direct pointers use the slot's back-pointer to access the object's indirection table entry which contains a pointer to its new memory slot. To improve the performance of future accesses to this object, the query also updates the direct pointer to the object's new memory location. The forwarding flag is set by the thread relocating the object after completing the relocation in the same atomic operation that unsets the frozen and lock bits; hence, tombstones cannot be reached through (indirect) references. As was the case for the two other flags, checking the forwarding flag is performed during incarnation number checking and, hence, does not penalize the common case of an unset forwarding flag.

Tombstoned memory slots are not reclaimed until there are no more direct pointers to them. After compacting an SMC, the compaction thread scans all SMCs that have direct pointers to it and updates the pointers to relocated objects. Note that the references between SMCs are statically known and the compiler can produce specialized functions that only scan SMCs that have direct pointers that may have to be updated and only check the corresponding pointer fields. We improve the performance of scanning an SMC to update direct pointers by only following pointers to memory slots that are known to have been relocated. This saves many random memory accesses. We achieve this by building a hash table during compaction that contains the memory addresses of all blocks that are compacted and, instead of following a direct pointer to see if the forwarding flag is set, we first compute the address of the corresponding block, probe it in the hash table and only follow the direct pointer if the block address was in the hash table.

7. EVALUATION

We implemented SMCs as a library using `unsafe` C♯ code. We did not change the JIT-compiler to automatically inject the code for correctly dereferencing references to self-managed objects but added this code by hand to factor out any overhead. We implemented the code generation techniques of [13] and we did not use any query-specific optimizations. Our experimental setup was an Intel Core i7-2700K (4x3.5GHz) system with 16GB of RAM, running Windows 8.1 and .NET 4.5.2. We compare SMCs with the default managed collection types in C♯. Unlike SMCs, most collections in C♯ are not thread-safe (*e.g.,* `List<T>`, C♯'s version of a dynamic array). Thread-safe collection types in C♯ are limited and only `ConcurrentDictionary<TKey, TValue>` and `ConcurrentBag<T>` provide comparable functionality to SMCs; however, `ConcurrentBag<T>` does not allow the removal of specific objects. .NET supports two garbage collection modes: *workstation* and *server*. Both modes support either interactive (concurrent) or batch (non-concurrent) garbage collections. In our tests the server modes outperformed the workstation ones, so we only report results for the server mode and only report both concurrency settings if their results differ.

Our benchmarks are primarily based on an object-oriented adaptation of the TPC-H workload. We have chosen to focus on a database benchmark as we believe it exemplifies the class of large-scale analytics applications that will benefit from SMCs. A relational workload is the most typical example of an application that has traditionally offloaded 'heavy' data-bound computation to an optimized runtime for that data model (a relational DBMS). As such, it is a good indication of both the classes of queries that can be integrated in the programming language, while, at the same time, it can provide an immediate performance comparison to the dominant alternative. TPC-H tables map to collections and each record to an object composed of C♯'s primitive types and references to other records (all primary-foreign-key relations). Based on the latter, most joins are performed using references. Unless stated otherwise, we use a scale factor of three for all TPC-H benchmarks. Note that due to a 16-byte-per-object overhead and larger primitive types (*e.g.,* `decimal` is 16 bytes wide) in C♯, a scale factor of three requires significantly more memory than in a database system.

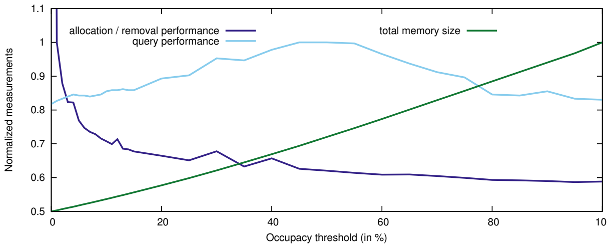


Figure 6: Varying the relocation threshold

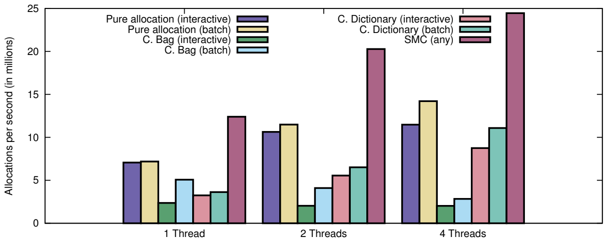


Figure 7: Batch allocation throughput

Sensitivity to relocation threshold Recall from §3.4 that the data blocks of SMCs may contain limbo slots that cannot be reclaimed yet and that we use a tolerance threshold of such slots in a block that needs to be surpassed before adding the block to a reclamation queue. Varying this threshold affects the memory size, the cost of memory operations and the query performance of SMCs. In Figure 6 we show how these factors change when varying the threshold (normalized to the maximum value). As the percentage of unused limbo slots grows, so does the memory footprint of the collection. The cost of performing memory operations (*i.e.,* insertions and removals) slowly decreases with an increasing threshold as allocations have to scan less memory slots to find a slot that can be reclaimed. Query performance seems to be less dependent on the additional slot directory entries that have to be processed with an increasing threshold, but more on the branch misprediction penalties when verifying if the slot is occupied. At a 50% threshold, the branch predictor has the most trouble predicting if the slot is occupied. Based on the results of Figure 6, we will use a 5% threshold for the following experiments. For a 5% threshold, the memory requirements of SMCs are comparable to that of storing managed objects in `List<T>`.

Memory allocation throughput In Figure 7 we compare the throughput (in objects per second) of allocating `lineitem` objects (using the default constructor) in an SMC to the pure allocation throughput of managed objects in

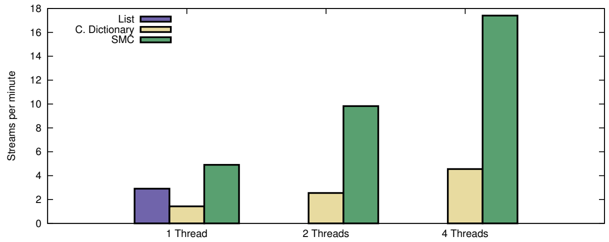


Figure 8: Refresh stream throughput

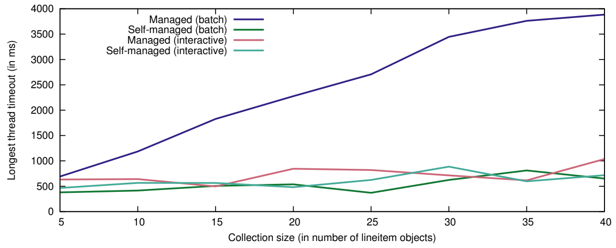


Figure 9: Timeouts caused by garbage collection

.NET[3] and the throughput of allocating managed objects and adding them to a concurrent collection. For managed allocations we report the throughput for interactive and batch garbage collection; the latter consistently provides better performance. SMCs significantly outperform both managed collections and the pure allocation cost of managed objects. All objects remain reachable so the runtime performs numerous garbage collections, with many of them stopping all application threads to copy objects from younger to older generations. SMCs allocate from (previously unused) thread-local blocks, which reduces the synchronization overhead of multiple allocation threads to about one atomic operation per 10k `lineitem` allocations.

Refresh streams To measure the throughput of memory operations we introduce the TPC-H refresh streams. Each thread continuously runs one of two kinds of streams with the same frequency. The first stream type creates and adds `lineitem` objects (0.1% of the initial population) to the `lineitem` collection. The second stream type enumerates all elements in the `lineitem` collection and removes 0.1% of the initial population based on a predicate on the object's `orderkey` value. All 0.1% objects to delete are provided in a hash map and removed in a single enumeration over the collection. This benchmark represents the common use case of refreshing the data stored in SMCs. In Figure 8 we report the stream throughput for SMCs against `ConcurrentDictionary<TKey, TValue>`; `ConcurrentBag<T>` is not included because it does not support the removal of specific elements. SMCs perform better than both types of managed collections in all cases.

Impact of garbage collection Out of the two garbage collection settings reported in Figure 7, the (non-concurrent) batch mode provides the higher throughput. In other garbage collection intensive benchmarks, we found the batch mode to enable a several times higher throughput. However, the

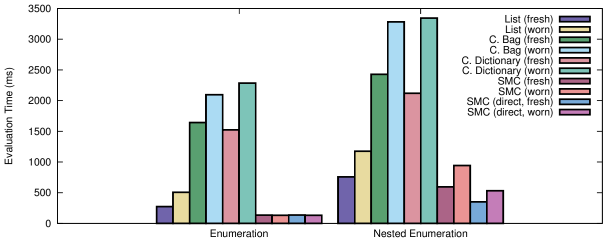


Figure 10: Enumeration performance

higher throughput comes at a price: response time. Where concurrent collectors (interactive) can perform big parts of garbage collection on a background thread without pausing all application threads, non-concurrent collectors have to pause all threads for the duration of the collection. As the size of the managed heap grows, so does the duration of full garbage collections and, hence, the application's maximum response time. To illustrate this, we insert a number of objects into a collection, either managed or self-managed, and then start two threads in parallel. The first thread continuously allocates managed objects with varying lifetimes and the second continuously sleeps for one millisecond and measures the time that passed in the meantime. If it observes that significantly more time has passed than expected, it records the value as it most likely was caused by garbage collection triggered by the other thread. Figure 9 shows the maximum timeout measured for a varying number of objects stored in the collection. For non-concurrent garbage collection, the maximum timeout increases with a growing number of objects stored in a managed collection, but remains fairly stable when these objects are stored in an SMC. Thus, the duration of garbage collections increases with growing data volumes stored in the managed heap. In the batch mode this negatively impacts the responsiveness of the application; in the interactive mode, it negatively impacts the overall application performance as the background collection thread steals processing resources from the application. In both cases, SMCs scale better with increasing data volumes.

Enumeration performance We first report on the pure enumeration performance of SMCs before considering more complex queries. Our queries either: (*a*) enumerate the `lineitem` collection and perform a simple function on each object to ensure that all `lineitem` objects are accessed; or (*b*) enumerate the `lineitem` collection, and for each object follow the `order` reference to a `customer` object and perform a simple function on the latter to ensure that `customer` objects are also accessed. Query performance deteriorates over time as objects are added and removed from the collection. In managed collections, objects may end up scattered all over the managed heap, whereas in SMCs the blocks containing objects may have holes due to limbo slots. In Figure 10 we show the performance of both query types after the collections are freshly loaded (fresh) and after the collections have undergone numerous object removals and insertions (worn). SMCs (indirect) outperform all automatically managed collections. However, when performing nested object accesses, the difference with `List<T>` diminishes because of the additional memory access required by indirection when following self-managed references. By utilizing the direct pointers of §6, we can bypass this look-up and

[3] Pre-allocated, thread-local arrays prevent objects from being garbage collected.

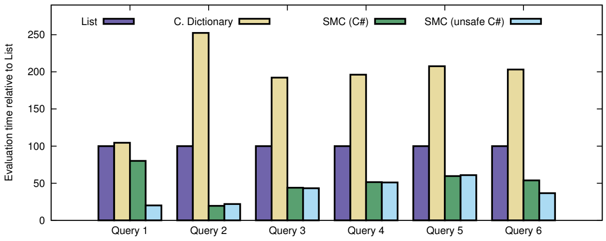


Figure 11: TPC-H Queries 1 to 6

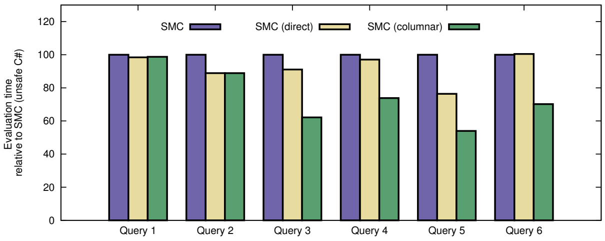


Figure 12: Direct pointer and columnar storage

improve performance. When comparing the fresh and worn states, SMCs only lose performance under nested accesses, whereas managed collections exhibit degraded performance in both cases. As `ConcurrentDictionary<TKey, TValue>` is the best performing thread-safe managed collection, we exclude `ConcurrentBag<T>` in what follows.

Query processing In Figure 11 we show the performance of the object-oriented adaptation of the first six TPC-H queries. For managed collections, we report the query performance of compiled C$^\sharp$ code (as in [13] but with reference-based joins). Using LINQ to evaluate the queries instead of compiling them to C$^\sharp$ code results in a 40% to 400% higher evaluation time, but as this was not the focus of the paper, we do not report it in Figure 11. We report on two versions of compiled code for SMCs: (*a*) Compiled C$^\sharp$ code that, other than the enumeration code, is equivalent to the code used for managed collections. This illustrates the fraction of the overall improvement contributed by the better enumeration performance of SMCs. (*b*) Compiled `unsafe` C$^\sharp$ code that contains optimizations only possible on SMCs. One such optimization is to use direct pointers to primitive types in an object (*e.g.,* `decimal` values) as arguments to functions that operate on them (*e.g.,* addition). For managed objects, these functions have to be called by value as the garbage collector may move the object inside the managed heap at any time without notice and, hence, the pointer would become invalid. Another optimization is to use memory regions [16] for all intermediate data during query processing, which improves performance by excluding those intermediates from garbage collection. Figure 11 reports the query processing performance relative to the performance of `List<T>`. SMCs perform significantly better than `ConcurrentDictionary<TKey, TValue>`, the fastest competing thread-safe collection in .NET; and even between 47% and 80% better than `List<T>`. Query 1 is a great example of what can be achieved with direct pointer access to self-managed objects. The query is `decimal` computation heavy and as C$^\sharp$'s `decimal` type is 16-bytes wide, calling the functions that perform decimal math using pointers and allowing for in-place modifications results in a huge performance gain. The other queries are less `decimal` computation intensive and, hence, show very little improvement from using unsafe code. Generating native C code leads to another 10% to 20% improvement over compiled `unsafe` C$^\sharp$ code. But as the compiled C code is (mostly) equivalent to the compiled `unsafe` C$^\sharp$ code, any performance differences can be attributed to more aggressive code-level optimizations by the C compiler.

Direct pointers and columnar storage In Figure 12 we show the impact of the direct pointer optimization introduced in §6 and columnar storage as discussed in §4.1.

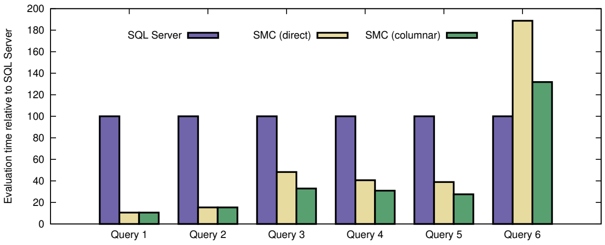


Figure 13: Comparison to SQL Server on a TPC-H-like workload

Direct pointer moderately improve query performance for queries that contain joins, in particular for Query 5. Columnar storage shows further improvements that are enabled by the SMCs decoupling the memory layout of their elements from their definition through managing their own memory.

Comparison to RDBMS To put the SMC results into perspective, we compare the query performance over objects in SMCs to that of a modern commercial database system. We use SQL Server 2014 for this purpose as it is well integrated into .NET and incorporates a compressed in-memory columnar store. We store all tables in the database's column store and, in addition, use clustered indexes on `shipdate` and `orderdate`. We use the `read uncommitted` isolation level and disable parallelized query execution to level the playing field. The results are shown in Figure 13. For most of the queries, SMCs exhibit better query performance. For join-heavy queries, they benefit from using references to perform joins instead of explicit value-based join operations. In other queries the database benefits from the indexes on `shipdate` and `orderdate`.

8. RELATED WORK

Type-safe manual memory management is at the core of SMCs. Region-based memory management [16] groups objects in regions and deallocates entire regions. Deallocating objects at region granularity is too high a storage overhead as objects in the applications we target are long-lived with only incremental insertions and deletions. Memory safety at object granularity is enforced by introducing specialized pointer types, *e.g., smart pointers* in C++11, which use reference counting to ensure that memory is only freed once it is no longer referenced. Reference counting comes at a high cost, especially when objects may be accessed concurrently [11]. *Fat pointers* are frequently used for type and/or memory safety at run-time [1]. Tracking object incarnations [6] is an application of this approach.

We use a variant of epoch-based memory reclamation used in lock-free data structures [5, 7], to ensure thread-safety. *Hazard pointers* and their variants [8, 11] ensure that threads only reclaim memory that is not referenced by other such pointers. This is similar to our epoch-based approach, but it would reduce performance: each query would iterate over objects through a hazard pointer, requiring a memory barrier whenever it is assigned to the next object. Epochs amortize the cost of memory barriers by using the entire query as the granularity of the critical section. Braginsky and Petrank [3] propose a lock-free sorted linked list optimized for spatial locality. Each list element is a sub-list of several data elements stored as a chunk of memory. Hazard pointers ensure safe memory reclamation, while a freeze bit in the elements' next pointer ensures lock-free splitting and merging of chunks. The implementation is limited to a specific format for each list element (`integer` key and value).

To improve query performance, SMCs rely on query compilation [9, 10, 14, 15]. We use popular techniques, *e.g.,* maximizing the processing performed in each loop and merging query operations inside a loop to maximize data reuse [14]. Klonatos et al. [9] propose the use of a high-level programming language for implementation and use query compilation for query processing. In contrast to our approach, the data store and query processor are not integrated with the application and, hence, the database functionality is treated as a black box (*e.g.,* there are no references to data objects). Murray et al. [12] first proposed query compilation for LINQ queries on in-memory objects. Their code generation approach did not go beyond querying C$^\sharp$ objects in managed collections using compiled C$^\sharp$ code. Nagel et al. [13] extended that idea by experimenting with different data layouts and identified managed collections as a performance bottleneck; generating native C code that operates on arrays of in-place structs provided the best performance. Our work builds on these findings.

DryadLINQ [17] and Trill [4] both build on LINQ to ease programming and to provide better application integration. DryadLINQ transforms LINQ programs into distributed computations running on a cluster whereas Trill operates on data batches pushed from external sources.

9. CONCLUSION

In this paper we introduced self-managed collections, a new type of collection for managed applications that manage and process large volumes of in-memory data. SMCs have specialized semantics that allow the collection to manually manage the memory space of its contained objects; and the objects of the collection to be referenced from the application and other SMCs. SMCs are optimized for query processing using language-integrated queries compiled to imperative code. We introduced the type-safe manual memory management system of SMCs and then the collection type itself. Our evaluation shows that SMCs outperform managed collections on query performance, batch allocations, and online modifications using predicate-based removal. At the same time, SMCs can improve the response time of the application overall by reducing the stress on the garbage collector and allow it to better scale with growing data volumes. Such scalability is transparent to the developer and eliminates the current required practise of resorting to low-level programming techniques.

10. REFERENCES

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI*, 1994.

[2] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *VLDB*, 2005.

[3] A. Braginsky and E. Petrank. Locality-conscious lock-free linked lists. In *ICDCN*. 2011.

[4] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. In *VLDB*, 2014.

[5] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *IEEE TOPADS*, 23(2):375–382, 2012.

[6] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *NSDI*, 2014.

[7] K. Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, 2004.

[8] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM TOCS*, 23:146–196, 2005.

[9] I. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. In *VLDB*, 2014.

[10] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[11] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *TPDS*, 2004.

[12] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *PLDI*, 2011.

[13] F. Nagel, G. Bierman, and S. D. Viglas. Code generation for efficient query processing in managed runtimes. In *VLDB*, 2014.

[14] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9), 2011.

[15] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled query execution engine using JVM. In *ICDE*, 2006.

[16] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and computation*, 1997.

[17] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

Lightweight Data Compression Algorithms: An Experimental Survey

Experiments and Analyses

Patrick Damme, Dirk Habich, Juliana Hildebrandt, Wolfgang Lehner
Database Systems Group
Technische Universität Dresden
01062 Dresden, Germany
{firstname.lastname}@tu-dresden.de

ABSTRACT

Lightweight data compression algorithms are frequently applied in in-memory database systems to tackle the growing gap between processor speed and main memory bandwidth. In recent years, the vectorization of basic techniques such as delta coding and null suppression has considerably enlarged the corpus of available algorithms. As a result, today there is a large number of algorithms to choose from, while different algorithms are tailored to different data characteristics. However, a comparative evaluation of these algorithms under different data characteristics has never been sufficiently conducted in the literature. To close this gap, we conducted an exhaustive experimental survey by evaluating several state-of-the-art compression algorithms as well as cascades of basic techniques. We systematically investigated the influence of the data properties on the performance and the compression rates. The evaluated algorithms are based on publicly available implementations as well as our own vectorized reimplementations. We summarize our experimental findings leading to several new insights and to the conclusion, that there is no single-best algorithm.

1. INTRODUCTION

The continuous growth of data volumes is a major challenge for the efficient data processing. This applies not only to database systems [1, 5] but also to other areas, such as information retrieval [3, 18] or machine learning [8]. With the growing capacity of the main memory, efficient analytical data processing becomes possible [4, 11]. However, the gap between computing power of the CPUs and main memory bandwidth continuously increases, which is now the main bottleneck for an efficient data processing. To overcome this bottleneck, data compression plays a crucial role [1, 22]. Aside from reducing the amount of data, compressed data offers several advantages such as less time spent on load and store instructions, a better utilization of the cache hierarchy, and less misses in the translation lookaside buffer.

This compression solution is heavily exploited in modern in-memory column stores for efficient query processing [1, 22]. Here, relational data is maintained using the *decomposition storage model* [6]. That is, an n-attribute relation is replaced by n binary relations, each consisting of one attribute and a surrogate indicating the record identity. Since the latter contains only *virtual* ids, it is not stored explicitly. Thus, each attribute is stored separately *as a sequence of values*. For the lossless compression of sequences of values (in particular integer values), a large variety of lightweight algorithms has been developed [1, 2, 3, 9, 12, 15, 16, 17, 18, 22][1]. In contrast to heavyweight algorithms like arithmetic coding [19], Huffman [10], or Lempel Ziv [21], lightweight algorithms achieve comparable or even better compression rates. Moreover, the computational effort for the (de)compression is lower than for heavyweight algorithms. To achieve these unique properties, each lightweight compression *algorithm* employs one or more basic compression *techniques* such as frame-of-reference [9, 22] or null suppression [1, 15], which allow the appropriate utilization of contextual knowledge like value distribution, sorting, or data locality.

In recent years, the efficient *vectorized* implementation of these lightweight compression algorithms using SIMD (Single Instruction Multiple Data) instructions has attracted a lot of attention [12, 14, 16, 18, 20], since it further reduces the computational effort. To better understand these vectorized lightweight compression algorithms and to be able to select a suitable algorithm for a given data set, the behavior of the algorithms regarding different data characteristics has to be known. In particular, the behavior in terms of *performance* (compression, decompression, and processing) and *compression rate* is of interest. In the literature, there are two papers with a considerable evaluation part. First, Adabi et al. [1] evaluated a small number of *unvectorized* algorithms on different data characteristics, but they neither considered a rich set of data distributions nor the explicit combination of different compression techniques. Second, Lemire et al. [12] already evaluated *vectorized* lightweight data compression algorithms, but considered only null suppression with and without differential coding. Furthermore, their focus is on postings lists from the IR domain, which narrows the considered data characteristics. Hence, an exhaustive comparative evaluation as a foundation has been never sufficiently conducted. To overcome this issue, we have

[1] Without claim of completeness.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.08

done an experimental survey of a broad range of algorithms with different data characteristics in a systematic way. In our evaluation, we used a set of synthetic data sets as well as one commonly used real data set. Our main findings can be summarized as follows:

1. Performance and compression rate of the algorithms vary greatly depending on the data properties. Even algorithms that are based on the same techniques, show a very different behavior.
2. By combining various basic techniques, the compression rate can be improved significantly. The performance may rise or fall depending on the combination.
3. There is no single-best lightweight algorithm, but the decision depends on the data properties. In order to select an appropriate algorithm, a compromise between performance and compression rate must be defined.

The remainder of the paper is organized as follows: In Section 2, we present more details about the area of lightweight data compression and introduce our evaluated algorithms. The implementation aspects are described in Section 3, while Section 4 covers our evaluation setup. Selected results of our experimental survey are presented in Section 5 and Section 6. Finally, we conclude the paper in Section 7.

2. PREREQUISITES

The focus of our experimental survey is the large corpus of *lossless lightweight integer data compression algorithms* which are heavily used in modern in-memory column stores [1, 22]. To better understand the algorithm corpus, this section briefly summarizes the basic concepts and introduces the algorithms which are used in our survey.

2.1 Lightweight Data Compression

First of all, we have to distinguish between *techniques* and *algorithms*, thereby each algorithm implements one or more of these techniques.

Techniques. There are five basic lightweight techniques to compress a sequence of values: frame-of-reference (FOR) [9, 22], delta coding (DELTA) [12, 15], dictionary compression (DICT) [1, 22], run-length encoding (RLE) [1, 15], and null suppression (NS) [1, 15]. FOR and DELTA represent each value as the difference to either a certain given reference value (FOR) or to its predecessor value (DELTA). DICT replaces each value by its unique key in a dictionary. The objective of these three well-known techniques is to represent the original data as a sequence of small integers, which is then suited for actual compression using the NS technique. NS is the most studied lightweight compression technique. Its basic idea is the omission of leading zeros in the bit representation of small integers. Finally, RLE tackles uninterrupted sequences of occurrences of the same value, so called *runs*. Each run is represented by its value and length. Hence, the compressed data is a sequence of such pairs.

Generally, these five techniques address different data levels. While FOR, DELTA, DICT, and RLE consider the *logical* data level, NS addresses the *physical* level of bits or bytes. This explains why lightweight data compression algorithms are always composed of one or more of these techniques. In the following, we also denote the techniques from the logical level as preprocessing techniques for the physical compression with NS. These techniques can be further divided into two groups depending on how the input values are mapped to output values. FOR, DELTA, and DICT map each input value to exactly one integer as output value (*1:1 mapping*). The objective of these preprocessing techniques is to achieve smaller numbers which can be better compressed on the bit level. In RLE, not every input value is necessarily mapped to an encoded output value, because a successive subsequence of equal values is encoded in the output as a pair of run value and run length (*N:1 mapping*). In this case, a compression is already done at the logical level. The NS technique is either a 1:1 or an N:1 mapping depending on the implementation.

Algorithms. The genericity of these techniques is the foundation to tailor the algorithms to different data characteristics. Therefore, a lightweight data compression algorithm can be described as a cascade of one or more of these basic techniques. On the level of the lightweight data compression algorithms, the NS technique has been studied most extensively. There is a very large number of specific algorithms showing the diversity of the implementations for a single technique. The pure NS algorithms can be divided into the following classes [20]: (i) bit-aligned, (ii) byte-aligned, and (iii) word-aligned.[2] While bit-aligned NS algorithms try to compress an integer using a minimal number of *bits*, byte-aligned NS algorithms compress an integer with a minimal number of *bytes* (1:1 mapping). The word-aligned NS algorithms encode as many integer values as possible into 32-bit or 64-bit words (N:1 mapping). The NS algorithms also differ in their data layout. We distinguish between *horizontal* and *vertical* layout. In the horizontal layout, the compressed representation of subsequent values is situated in subsequent memory locations. In the vertical layout, each compressed representation is stored in a separate memory word.

The logical-level techniques have not been considered to such an extent as the NS technique *on the algorithm level*. In most cases, the preprocessing steps have been investigated in connection with the NS technique. For instance, PFOR-based algorithms implement the FOR technique in combination with a bit-aligned NS algorithm [22]. These algorithms usually subdivide the input in subsequences of a fixed length and calculate two parameters per subsequence: a reference value for the FOR technique and a common bit width for NS. Each subsequence is encoded using their specific parameters, thereby the parameters are data-dependently derived. The values that cannot be encoded with the given bit width are stored separately with a greater bit width.

2.2 Considered Algorithms

We consider all five basic lightweight techniques in detail. Regarding the selected algorithms, we investigate both, implementations of a single technique as well as cascades of one logical-level and one physical-level technique. We decided to reimplement the logical-level techniques on our own (see Section 3) in order to be able to freely combine them with all seven considered NS algorithms (see Table 1). In the following, we briefly sketch each considered NS algorithm.

2.2.1 *Bit-Aligned NS Algorithms*

4-Gamma Coding [16] processes four input values (data elements) at a time. All four values are stored in the vertical storage layout using the number of bits required for the

[2] [20] also defines a *frame-based* class, which we omit, as the representatives we consider also match the *bit-aligned* class.

largest of them. The unary representation of the bit width is stored in a separate memory area for decompression.

SIMD-BP128 [12] processes data in blocks of 128 integers at a time. All 128 integers in the block are stored in the vertical layout using the number of bits required for the largest of them. The used bit width is stored in a single byte, whereby 16 of these bit widths are followed by 16 compressed blocks.

SIMD-FastPFOR [12] is a variant of the original PFOR algorithm [22], whose idea is to classify all data elements as either regular coded values or exceptions depending on if they can be represented with a certain bit width. This bit width is chosen such that the overall compression rate becomes optimal. All data elements are packed with the chosen bit width using the vertical layout. The exceptions require a special treatment, since that number of bits does not suffice for them. In SIMD-FastPFOR the exceptions are stored in additional packed arrays. The overall input is subdivided into pages which are further subdivided into blocks of 128 integers. SIMD-FastPFOR stores the exceptions at the page level and uses an individual bit width for each block.

2.2.2 Byte-Aligned NS Algorithms

4-Wise Null Suppression [16] compresses integers by omitting leading zero bytes. For each 32-bit integer, between zero and three bytes might be omitted. 4-Wise NS processes four data elements at a time and combines the corresponding four 2-bit descriptors into a 1-byte mask. In the output, four masks are followed by four compressed blocks in the horizontal layout.

Masked-VByte [14] uses the same compressed representation as the VByte algorithm [12] and differs only in implementation details. It subdivides an integer into 7-bit units. Each unit that is required to represent the integer produces one byte in the output. The seven data bits are stored in the lower part of that byte, while the most significant bit is used to indicate whether or not the next byte belongs to the next data element. Subsequent compressed values are stored using the horizontal layout.

2.2.3 Word-Aligned NS Algorithms

Simple-8b [2] outputs one compressed block of 64 bits for a variable number of uncompressed integers. Within one block, all data elements are stored with a common bit width using the horizontal layout. The bit width is chosen such that as many subsequent input elements as possible can be stored in the compressed block. One compressed block contains 60 data bits and a 4-bit selector specifying the compression mode. There are 16 compression modes: 60 1-bit values, 30 2-bit values, 20 3-bit values, and so on. Additionally, Simple-8b has two special modes indicating that the input consisted of 120 respectively 240 zeroes.

SIMD-GroupSimple [20] processes the input in units of so-called *quads*, i.e., four values at a time. For each quad, it determines the number of bits required for the largest element. Based on the bit widths of subsequent quads, it partitions the input sequence into groups, such that as many quads as possible can be stored in *four* consecutive 32-bit words using the vertical layout. There are ten compression modes: the four consecutive 32-bit words could be filled with 4 × 32 1-bit values, 4 × 16 2-bit values, 4 × 10 3-bit values, and so on. A four bit selector represents the mode chosen for the compressed block. The selectors are stored in a different memory area than the compressed blocks.

Class	Algorithm	Layout	Code origin	SIMD
bit-aligned	4-Gamma	vert.	Schlegel et al.	yes
	SIMD-BP128	vert.	FastPFor-lib	yes
	SIMD-FastPFOR	vert.	FastPFor-lib	yes
byte-aligned	4-Wise NS	horiz.	Schlegel et al.	yes
	Masked-VByte	horiz.	FastPFor-lib	n/y
word-aligned	Simple-8b	horiz.	FastPFor-lib	no
	SIMD-GroupSimple	vert.	our own code	yes

Table 1: The considered NS algorithms.

3. IMPLEMENTATION ASPECTS

As already mentioned, we reimplemented all four logical-level techniques in C++, i.e., DELTA, DICT, FOR, and RLE. Regarding the physical-level, several high-quality open-source implementations of NS are available. We used these existing implementations whenever possible and reimplemented only one of them. Table 1 summarizes the origins of the implementations we employed. We also implemented cache-conscious generic cascades of logical-level techniques and NS. Furthermore, we implemented a decompression with aggregation for all algorithms to evaluate a processing of compressed data. In this section, we describe some of the most crucial implementation details with respect to performance.

3.1 SIMD Instruction Set Extensions

Single Instruction Multiple Data (SIMD) instruction set extensions such as Intel's SSE and AVX have been available in modern processors for several years. SIMD instructions apply one operation to multiple elements of so-called *vector registers* at once. The available operations include parallel arithmetic, logical, and shift operations as well as permutations. These are highly relevant to lightweight compression algorithms. In fact the main focus of recent research [12, 14, 16, 18, 20] in this field has been the employment of SIMD instructions to speed up (de)compression. Consequently, most algorithms we evaluate in this paper make use of SIMD extensions (see Table 1). Vectorized load and store instructions can be either aligned or unaligned. The former require the accessed memory addresses to be multiples of 16 bytes (SSE) and are usually faster. Although nowadays Intel's AVX2 offers 256-bit operations, we decided to restrict our evaluation to implementations using 128-bit SSE. This has two reasons: (1) Most of the techniques presented in the literature are designed for 128-bit vector operations[3] and (2) The comparison is fairer if only one width of vector registers is considered. Intel's SIMD instructions can be used in C/C++ without writing assembly code via intrinsic functions, whose names start with `_mm_`.

3.2 Physical-Level Technique: NS

In the following, we describe crucial points regarding existing implementations as well as one reimplementation.

3.2.1 Bit-Aligned Algorithms

We obtained the implementation of **4-Gamma Coding** directly from the authors [16], and those of **SIMD-BP128** and **SIMD-FastPFor** from the FastPFor-library [13]. All

[3] Thereby, a transition to 256-bit operations is not always trivial and could be subject to future research.

three implementations use vectorized shift and mask operations. SIMD-BP128 and SIMD-FastPFor use a dedicated optimized packing and unpacking routine for each of the 32 possible bit widths. It is worth mentioning, that – while the original PFOR algorithm is a combination of the FOR and the NS technique – SIMD-FastPFOR, despite its name, does not include the FOR technique, but only the NS technique.

3.2.2 *Byte-Aligned Algorithms*

Regarding **4-Wise Null Suppression**, we use the original implementation by Schlegel et al. [16]. It implements the horizontal packing of the uncompressed values using a vectorized byte permutation. The 16-byte permutation masks required for this are built once in advance and looked up from a table during the compression. This table is indexed with the 1-byte compression masks, thus there are 256 permutation masks in total. The decompression works by using the inverse permutation masks.

Masked-VByte vectorizes the decompression of the compressed format of the original VByte algorithm. The implementation we use is available in the FastPFor-library [13] and is based on code by the original authors. The crucial point of the vectorization is the execution of a SIMD byte permutation in order to reinsert the leading zero-bytes removed by the compression. After 16 bytes of compressed data have been loaded into a vector register, the most significant bits of all bytes are extracted using a SIMD instruction. The lower 12 bits of this 16-bit mask are used as a key to lookup the required permutation mask in a table. After the permutation, the original 7-bit units need to be stitched together, which is done using vectorized shift and mask operations. Masked-VByte also has an optimization for the case of 12 compressed 1-byte integers.

3.2.3 *Word-Aligned Algorithms*

We use the implementation of **Simple-8b** available in the FastPFor-library [13], which is a purely sequential implementation. It uses a dedicated sequential packing routine for each of the possible selectors.

We reimplemented **SIMD-GroupSimple** based on the description in the original paper, since we could not find an available implementation. We employed the two optimizations discussed by the original authors: (1) We calculate the pseudo-quad max values instead of the quad max values to reduce the number of branch instructions. (2) We use one dedicated and vectorized packing routine for each selector, whereby the correct one is chosen by a switch-statement.

The original compression algorithm processes the input data in *three* runs: The first run scans the entire input and materializes the pseudo-quad max array in main memory. The size of this array is one quarter of the input data size. The second run scans the pseudo-quad max array and materializes the selectors array. The third run iterates over the selectors array and calls the respective packing routine to do the actual compression. This procedure results in a suboptimal cache utilization, since at the end of each run, the data it started with has already been evicted from the caches. Thus, reaccessing it in the next run becomes expensive.

In order to overcome this issue, we enhanced the compression part of the algorithm with one more optimization, which was not presented in the original paper: Our reimplementation stores the pseudo-quad max values in a ring buffer of a small constant size (32 32-bit integers) instead of an array proportional to the input size. This is based on the observation that the decision for the next selector can never require more than 32 pseudo-quad max values, since at most 4×32 (1-bit) integers can be packed into four 32-bit words. Due to its small size (128 bytes), the ring buffer fits into the L1 data cache and can thus be accessed at top-speed. Our modified compression algorithm repeats the following steps until the end of the input is reached (in the beginning, the ring buffer is empty): (1) Fill the ring buffer by calculating the next up to 32 pseudo-quad max values. This reads up to $4 \times 32 = 128$ uncompressed integers. (2) Run the original subroutine for determining the next selector *on the ring buffer.* (3) Store the obtained selector to the selectors section in the output. (4) Compress the next block using the subroutine belonging to the selector. This will typically reread the uncompressed data touched in Step 1. Note that this data is very likely to still reside in the L1 cache, since only a few bytes of memory have been touched in between. (5) Increase the position in the ring buffer by the number of input quads compressed in the previous step. We observed that using this additional optimization, the compression part of our reimplementation is always faster than without it. Note, that this optimization does not affect the compressed output in any way.

3.3 Logical-Level Techniques

As previously mentioned, logical-level techniques are usually combined with NS in existing algorithms and are thus hardly available in isolation. In order to be able to freely combine *any* logical-level technique with *any* NS algorithm, we reimplemented all four logical-level compression techniques as stand-alone algorithms. Thereby, an important goal is the vectorization of those algorithms.

3.3.1 *Vectorized DELTA*

Our implementation of DELTA represents each input element as the difference to its fourth predecessor. This allows for an easy vectorization by processing four integers at a time. The first four elements are always copied from the input to the output. During the compression, the next four differences are calculated at once using `_mm_sub_epi32()`. The decompression reverses this by employing `_mm_add_epi32()`. This implementation follows the description in [12] with the difference that we do not overwrite the input data, because we still need it as the input for the other algorithms.

3.3.2 *Sequential DICT*

Our implementation of DICT is a purely sequential single-pass algorithm employing a static dictionary, which is built on the uncompressed data before the (de)compression takes place. Thus, building the dictionary is not included in our time measurements and the dictionary itself is not included in the compressed representation. This represents the case of a domain-specific dictionary which is known in advance. The compression uses a C++-STL `unordered_map` to map values to their keys, whereas the decompression uses the key as the index of a `vector` to look up the corresponding value.

3.3.3 *Vectorized FOR*

We implemented the compression of FOR as a vectorized two-pass algorithm. The first pass iterates over the input and determines the reference value, i.e., the minimum using `_mm_min_epu32()`. This minimum is then copied into all four elements of one vector register. The second pass iterates over the input again and subtracts this vector register from four input elements at a time using `_mm_sub_epi32()`. In the end, the reference value is appended to the output. The decompression adds this reference value to four data elements at a time using `_mm_add_epi32()`.

3.3.4 *Vectorized RLE*

Our implementation of RLE also utilizes SIMD instructions. The compression part is based on parallel comparisons. It repeats the following steps until the end of the input is reached: (1) One 128-bit vector register is loaded with four copies of the current input element. (2) The next four input elements are loaded. (3) The intrinsic `_mm_cmpeq_epi32()` is employed for a parallel comparison. The result is stored in a vector register. (4) We obtain a 4-bit comparison mask using `_mm_movemask_ps()`. Each bit in the mask indicates the (non-)equality of two corresponding vector elements. The number of trailing one-bits in this mask is the number of elements for which the run continues. If this number is 4, then we have not seen the run's end yet, and continue at step 2. Otherwise, we have reached the run's end and append the run value and run length to the output and continue with step 1 at the next element after the run's end.

The decompression executes the following until the entire input has been consumed: (1) Load the next pair of run value and run length. (2) Load one vector register with four copies of the run value. (3) Store the contents of that register to memory as often as required to match the run length.

3.4 Cascades of Logical-Level and Physical-Level Techniques

The challenge of implementing cascades, i.e., combinations of logical-level and physical-level techniques, is the high implementation effort due to the high number of possible combinations. To address this problem, we implemented a cache-conscious cascade which is generic w.r.t. the employed algorithms. That is, it can be instantiated for *any* two algorithms, without further implementation effort. It takes three parameters: a logical-level algorithm L, a physical-level algorithm P, and an (uncompressed) block size bs_u.

The output consists of compressed blocks, each of which starts with its size as a 32-bit integer followed by 12 bytes of padding to achieve the 16-byte alignment required by SSE instructions. The body of the block contains the compressed data possibly followed by additional padding bytes.

The compression procedure repeats the following steps until the end of the input is reached: (1) Skip 16 bytes in the output buffer. (2) Apply the compression of L to the next bs_u elements in the input. Store the result in an intermediate buffer. (3) Apply the compression of P to that buffer and store the result to the output buffer. (4) Store the size bs_c of the compressed block to the bytes skipped in Step 1. (5) Skip some bytes after the compressed block, if it is necessary to achieve 16-byte alignment.

The decompression is the reverse procedure repeatedly executing the following steps: (1) Read the size bs_c of the current compressed block and skip the padding. (2) Apply the decompression of P to the next bs_c bytes in the input. Store the result to an intermediate buffer. (3) Decompress the contents of that buffer using L and append the result to the output. (4) Skip the padding in the input, if necessary.

The intermediate buffer is reused for all blocks. Its size is in the order of magnitude of bs_u (we chose $4KiB + 2 \times bs_u$ as a pessimistic estimation). This algorithm is cache-conscious, if bs_u is chosen to fit the Lx cache, since then, the data read by the second algorithm is likely to still reside in that cache.

3.5 Decompression with Aggregation

We also modified the decompressions of both, our own reimplementations and existing implementations, such that they sum up the decompressed data instead of writing it to memory. The usual case for the vectorized algorithms is that four decompressed 32-bit integers reside in a vector register before they are stored to memory using `_mm_store_si128()`. We replaced these store instructions by vectorized additions. However, since the sum might require more than 32 bits, we first distribute the four 32-bit elements to the four 64-bit elements of *two* 128-bit registers using `_mm_unpacklo_epi32()` and `_mm_unpackhi_epi32()` and add both to *two* 64-bit running sums (which are added in the very end) by applying `_mm_add_epi64()`. In the case of RLE, we add the product of the run length and the run value to the running sum.

4. EVALUATION SETUP

In this section, we describe our overall evaluation setup. All algorithms are implemented in C/C++ and we compiled them with `g++` 4.8 using the optimization flag `-O3`. As the operating system we used Ubuntu 14.04. All experiments have been executed on the same hardware machine in order to be able to compare the results. The machine was equipped with an Intel Core i7-4710MQ (Haswell) processor with 4 physical and 8 logical cores running at 2.5 GHz. The L1 data, L2, and L3 caches have a capacity of 32 KB, 256 KB and 6 MB, respectively. We use only one core at any time of our evaluation to avoid competition for the shared L3 cache. The capacity of the DDR3 main memory was 16 GB. We are able to copy data using `memcpy()` at a rate of 6.15 GiB/s or 1,650 mis (million integers per second).

All experiments happened entirely in main memory. The disk was never accessed during the time measurements. The whole evaluation is performed using our benchmark framework [7]. The synthetic data generation was performed by our data generator once per configuration of data properties. The data properties were recorded and the algorithms were repeatedly performed on the generated data. During the executions, the runtimes and the compression rates were measured. Furthermore, we emptied the cache before each algorithm execution by copying a 12-MB array (twice as large as the L3 cache) using a loop operation.

All time measurements were carried out by means of the wallclock-time (C++-STL `high_resolution_clock`) and were repeated 12 times to receive stable values, thereby we only report average values. The time measurements include:

Compression: Loading uncompressed data from main memory, applying the compression algorithm, storing the compressed data to main memory

Decompression: Loading compressed data from main memory, applying the decompression algorithm, storing the uncompressed data to main memory

Decompression & Aggregation: Loading compressed data from main memory, applying the decompression and summation, storing 8 bytes in total to main memory

5. EXPERIMENTS ON SYNTHETIC DATA

In this section, we present selected results of our experi-

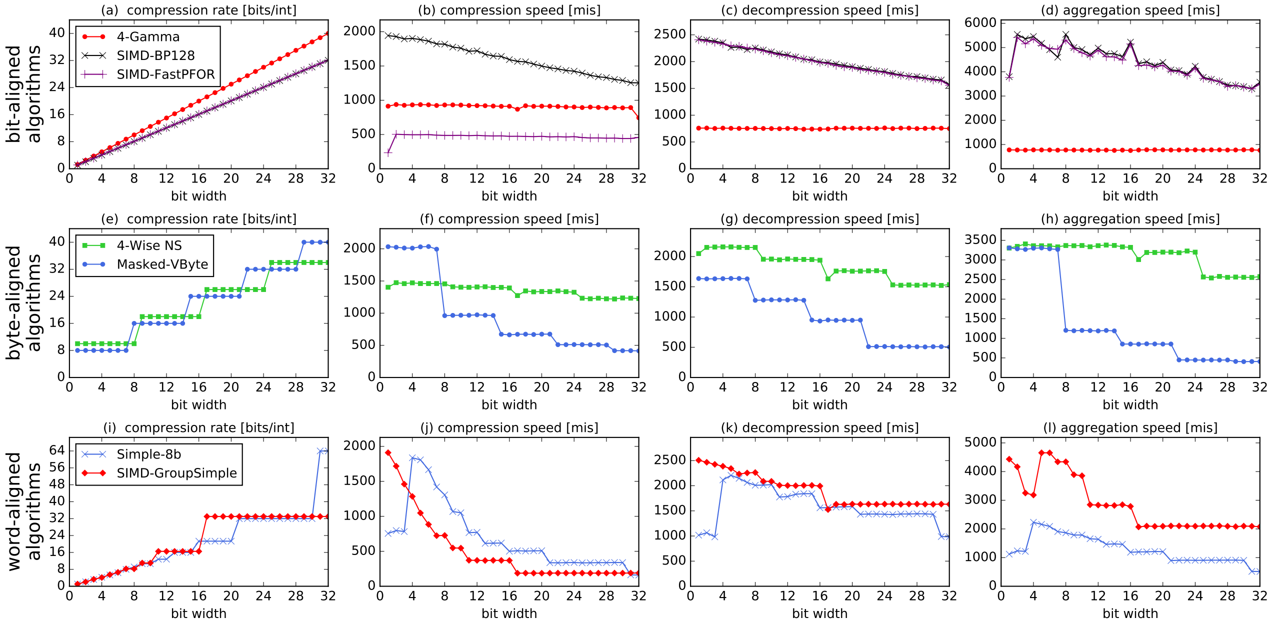


Figure 1: The general behavior of the three classes of NS algorithms.

mental survey. We generate synthetic data sets in order to be able to control the data properties in a systematic way. All uncompressed arrays contain 100 million 32-bit integers, i.e., 400 MB. Thus, only a small portion of the uncompressed data fits into the L3 cache. We report speeds in *million integers per second (mis)* and compression rates in *bits per integer (bits/int)*. We begin with the evaluation of pure NS algorithms in Section 5.1. After that, we investigate pure logical-level algorithms in Section 5.2. In Section 5.3, we evaluate cascades of logical-level techniques and NS. Finally, in Section 5.4 we present the conclusions we draw from our evaluation on synthetic data.

5.1 Null Suppression Algorithms

We start by identifying the characteristics of the three classes of NS algorithms. After that, we compare five selected NS algorithms in more detail.

5.1.1 Classes of NS Algorithms

We generate 32 unsorted datasets, such that all data elements in the i-th dataset have exactly i effective bits, i.e., the value range is $[0, 1]$ for $i = 1$ and $[2^{i-1}, 2^i)$ for $i = 2, \ldots, 32$. Within these ranges, the values are uniformly distributed.

Figure 1 (a-d) show the results for the considered **bit-aligned algorithms**. These have the finest possible compression granularity and thus can perfectly adapt to any bit width. Consequently, the compression rate is a linear function of the bit width. The speeds of compression, decompression, and aggregation follow the same linear trend. Nevertheless, there are differences between the algorithms. Since SIMD-BP128 and SIMD-FastPFOR store less meta information than 4-Gamma, they achieve better compression rates. They are also better regarding the decompression and aggregation speed. However, in terms of compression speed, SIMD-FastPFOR is by far the slowest, while SIMD-BP128 still shows very good performance.

Figure 1 (e-h) present the results for the considered **byte-aligned** algorithms. These algorithms compress an integer at the granularity of units of 8 bits (4-Wise NS) or 7 bits (Masked-VByte). As a result, the curves of all four measured variables exhibit a step-shape, whereby the step width is constant and equals the unit size of the algorithm. Since the units of 4-Wise NS and Masked-VByte have different sizes, the first and second regarding compression rate change several times when increasing the bit width. Concerning the speeds, 4-Wise NS is always at least as fast as Masked-VByte, except for the compression of values with up to 7 bits, in this case Masked-VByte is significantly faster.

Finally, Fig. 1 (i-l) provide the results for the **word-aligned** algorithms. These can adapt only to certain bit widths, which are not the multiples of any unit size. For instance, SIMD-GroupSimple supports 1, 2, 3, 4, 5, 6, 8, 10, 16, and 32 bits. Hence, the measured variables show steps at these bit widths, i.e., the steps do not have a constant width per algorithm. This basic shape is especially clear for the compression rate and the compression speed, but can also be found in the decompression and aggregation speed. SIMD-GroupSimple compresses better for certain bit widths, and for others, Simple-8b does. Since Simple-8b uses 64-bit words in the output, it can still achieve a size reduction for bit widths up to 20, while SIMD-GroupSimple cannot reduce the size anymore if the bit width exceeds 16. SIMD-GroupSimple is faster for bit widths up to 3 and slower in all other situations. However, regarding decompression and aggregation, it is faster for all bit widths.

To summarize, each of the three classes exhibits its individual behavior subject to the bit width. At the same time the differences between the classes are significant.

5.1.2 Detailed Comparison of NS Algorithms

For the following experiments we pick SIMD-BP128, SIMD-FastPFOR, 4-Wise NS, Masked-VByte, and SIMD-GroupSimple and investigate their behavior in more detail. Note that all three classes of NS are represented in this selection.

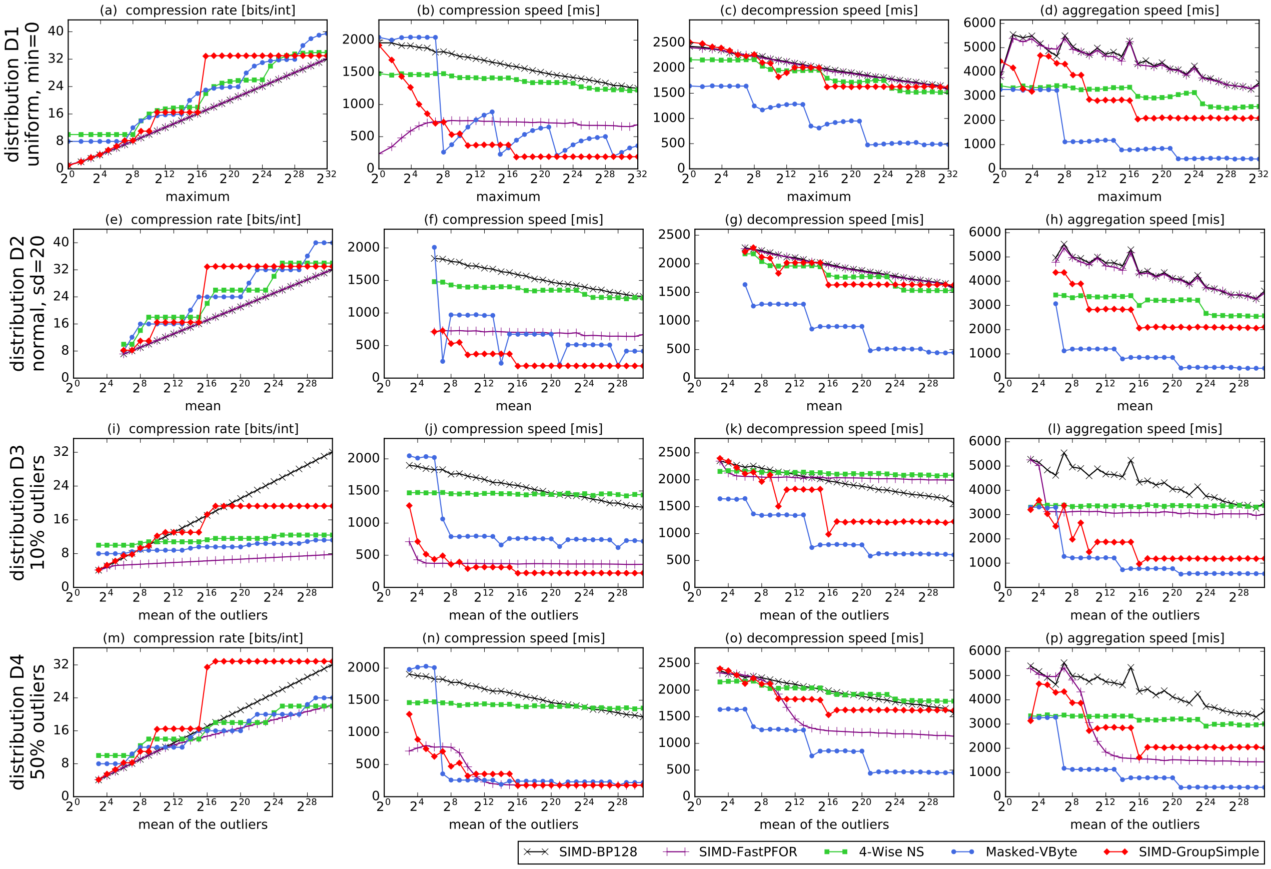


Figure 2: Comparison of NS algorithms of different classes on different data distributions.

We generate unsorted data using four distributions D1–4, whereby we vary one parameter for each of them. D1 is a uniform distribution with a min of 0 and a max varying from 0 to $2^{32} - 1$. D2 is a normal distribution with a standard deviation of 20 and a mean varying from 64 to 2^{31}. For D3, 90% of the values follow a normal distribution with a standard deviation of 2 and a mean of 8, while 10% are drawn from a normal distribution with the same standard deviation and a mean varying from 8 to 2^{31}. That is, 90% of the data elements are small integers, while 10% are increasingly large outliers. D4 is like D3, but with a ratio of 50:50. While D1–2 have a high data locality, D3–4 do not.

The results for D1 can be found in Fig. 2 (a-d). The bit-aligned algorithms SIMD-BP128 and SIMD-FastPFOR always achieve the best compression rates, since they can adapt to any bit width. Masked-VByte is the fastest compressor for small values, although it is not even vectorized. However, for larger values, SIMD-BP128 is the fastest, but comes closer to 4-Wise NS as the values grow. SIMD-GroupSimple yields the highest decompression speed for maximums up to 32. From there on SIMD-BP128 and SIMD-FastPFOR are the fastest, while SIMD-GroupSimple and 4-Wise NS come quite close to their performance, especially for the values for which they do not waste too many bits due to their coarser granularity.

For D2 (Fig. 2 (e-h)) we can make the same general observations. However, the steps in the curves of the byte-aligned algorithms become steeper, since D2 produces values with less distinct bit widths than D1.

The results of D3 (Fig. 2 (i-l)) reveal some interesting effects. Regarding the compression rate, SIMD-FastPFOR stays the winner, while SIMD-BP128 is competitive only for small outliers. For large outliers it even yields the worst compression rates of all five algorithms. This is due to the fact that SIMD-BP128 packs blocks of 128 integers with the bit width of the largest element in the block, i.e., one outlier per block affects the compression rate significantly. SIMD-FastPFOR on the other side, can handle this case very well, since it – like all variants of PFOR – is explicitly designed to tolerate outliers. The byte-aligned algorithms 4-Wise NS and Masked-VByte are worse than SIMD-FastPFOR, but still quite robust, since they choose an individual byte width for each data element and are, thus, not affected by outliers. SIMD-GroupSimple compresses better than SIMD-BP128 in most cases, since outliers lead to small input blocks, while there can still be large blocks of non-outliers. In terms of compression speed, SIMD-BP128 is still in the top-2, but it is overtaken by 4-Wise NS for large outliers. Concerning decompression speed, 4-Wise NS overtakes SIMD-BP128 when the outliers need more than 12 bits. SIMD-FastPFOR is nearly as fast as 4-Wise NS, but achieves much better compression rates. Regarding the aggregation, SIMD-BP128 is still the fastest algorithm, although SIMD-FastPFOR comes very close for small outliers and 4-Wise NS for large outliers.

D4 increases the amount of outliers to 50%. The compression rate of SIMD-BP128 does not change any more, since

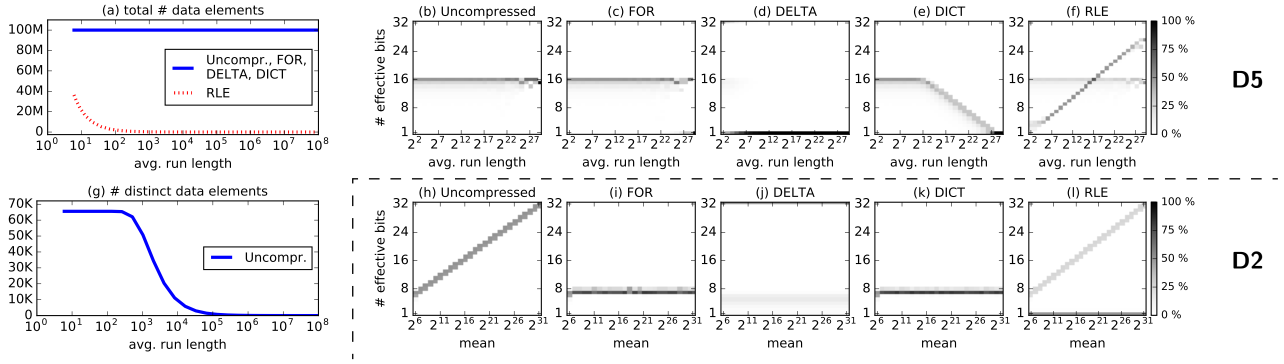


Figure 3: Logical-level techniques applied to D5 (a-g) and D2 (h-l): Data properties.[4]

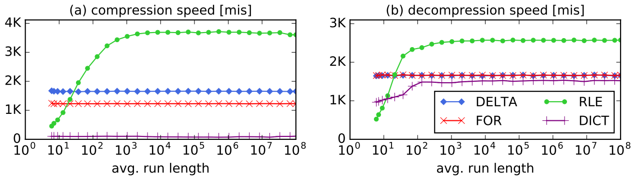


Figure 4: Logical-level techniques on D5: Speeds.

basically all blocks were affected by outliers in D3 already. However, since the other algorithms compress worse now, the trade-offs have to be reevaluated. Thanks to patched coding, SIMD-FastPFOR still is in the top-2 regarding the compression rate. However, this comes at the cost of (de)-compression and aggregation performance, which heavily decreases as the outliers grow. Encoding each value individually 4-Wise NS and Masked-VByte come very close to the compression rate of SIMD-FastPFOR and 4-Wise NS decompresses faster than SIMD-FastPFOR for large outliers.

To sum up, the best algorithm regarding compression rate or performance depends on the data distribution. Regarding one measured variable, a certain algorithm can be the best for one distribution and the worst for another distribution. Moreover, for a certain distribution the best algorithm regarding one measured variable can be the worst for another variable. In addition, there are many points of intersection between the algorithms' compression rates and speeds offering many different trade-offs.

5.2 Logical-Level Techniques

A general trend observable in Figures 1 and 2 is that all NS algorithms get worse as the data elements get larger. Logical-level techniques can be able to change the data properties in favor of NS. To illustrate this, we provide the results of the application of the four logical techniques to two *unsorted* datasets: D2, already known from the previous section, and D5, whose data elements are uniformly drawn from the range $[0, 2^{16} - 1]$ while varying the average run length.

We start with the discussion of D5. First of all, in Fig. 3 (a) we can see that the total number of data elements after the application of FOR, DELTA, and DICT is the same as in the uncompressed data (1:1 mapping), while with RLE it decreases significantly as the run length increases (N:1 mapping). This has two consequences: (1) an NS algorithm applied after RLE needs to compress less data and (2) RLE alone suffices to reduce the data size. Figure 3 (b-f)[4] show the data distributions in the uncompressed data as well as in the outputs of the logical-level techniques. Most uncompressed values have 16 or 15 effective bits. This does not change much with FOR, since the value distribution can produce values close to zero. In contrast, the output of DELTA contains nearly only values of one effective bit for long runs, since these yield long sequences of zeros. Note that there are also outliers having 32 effective bits, resulting from negative differences being represented in the two's complement. With DICT, the values start to get smaller as soon as the run length is high enough to lead to a decrease of the number of distinct values (see Fig. 3 (g)), and thus the maximum key. For RLE there are always two peaks in the distributions: one is at a bit width of 16 and corresponds to the run values and the other one is produced by the increasingly high run lengths. Note that this distribution is quite similar to D4 from the previous section. The distributions might seem to get worse for high run lengths. However, it must be kept in mind that RLE reduces the total number of data elements in those cases. Figure 4 provides the (de)compression speeds. The performance of DELTA and FOR is independent of the data characteristics, since they execute the same instructions for each group of four values. On the other side, RLE is slow for short runs, but becomes by far the fastest algorithm for long runs, since it has to write(read) less data during the (de)compression. DICT is the slowest compressor due to the expensive look ups in the map. Regarding the decompression, it is competitive to DELTA and FOR, but sensitive to the number of distinct values, which influences whether or not the dictionary fits into the Lx cache.

The distributions for D2 are visualized in Fig. 3 (h-l). Here, FOR can improve the distribution significantly, since the value range is narrow. The same applies to DICT, since consequently the number of distinct values is small. As the data is unsorted and does not have runs, about half of the values in the output of DELTA have 32 effective bits, i.e., the distributions get worse in most cases. Note that RLE doubles the number of data elements due to the lack of runs.

To sum up, logical-level techniques can significantly im-

[4] How to read Fig. 3 (b-f) and (h-l): The y-axis lists all possible numbers of effective bits a data element could have. Each vertical slice corresponds to one configuration of the data properties. The intensity encodes what portion of the data elements has how many effective bits. That is, the dark pixels show which numbers of effective bits occur most frequently in the dataset.

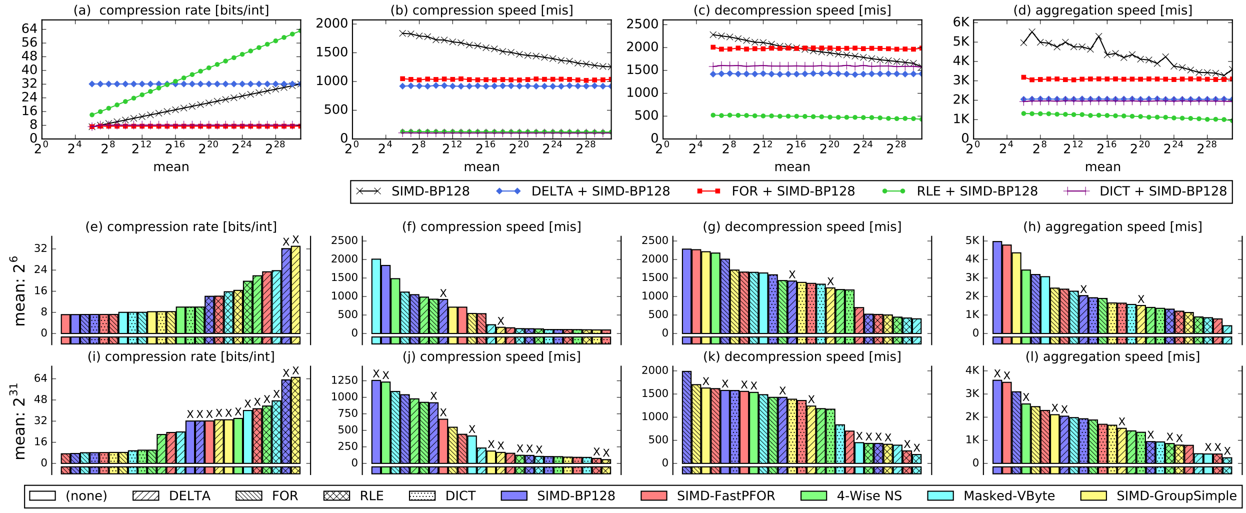


Figure 5: Comparison of the cascades on dataset D2.[5]

prove the data distribution in favor of NS. However, the data properties determine which techniques are suitable. In the worst case, the distributions might even become less suited. We also experimented with other data characteristics such as the number of distinct values and sorted datasets, but omit their results due to a lack of space. Those experiments lead to similar conclusions.

5.3 Cascades of Logical-Level and Physical-Level Techniques

To find out which improvements over the stand-alone NS algorithms the additional use of logical-level techniques can yield, we compare the five stand-alone NS algorithms from Section 5.1.2 to their cascades with the four logical-level techniques. That is, we compare $5 + 5 \times 4 = 25$ algorithms in total. The evaluation is conducted on three datasets: D1 and D5, which are already known, and D6, a *sorted* dataset for which we vary the number of distinct data elements by uniformly drawing values from the range $[0, max]$, whereby max starts with 0 and is increased until we reach 100 M distinct values, i.e., until all data elements are unique. For all three datasets, we provide a detailed comparison of SIMD-BP128 to its cascaded derivatives as well as a comparison of all 25 algorithms for selected data configurations. For our generic cascade algorithm, we chose a block size of 16 KiB, i.e., 4096 uncompressed integers. This size is a multiple of the block sizes of all considered algorithms and fits into the L1 cache of our machine. We also experimented with larger block sizes, but found that 16 KiB yields the best speeds.

Figure 5 (a-d) show the results of SIMD-BP128 and its cascaded variants on D2. The results for the compression rate are consistent with the distributions in Fig. 3 (h-l): Combined with FOR or DICT, SIMD-BP128 always yields equal or better results than without a preprocessing, while DELTA and RLE affect the results negatively. However, the cascades with logical-level techniques decrease the speeds of the algorithm, whereby the slow-down is significant for small data elements, but becomes acceptable for large values at least for DICT (decompression) and FOR. Indeed, the decompression of FOR + SIMD-BP128 is faster than SIMD-BP128 alone for means larger than 2^{16}. A comparison of all 25 algorithms can be found in Fig. 5 (e-h) and (i-l) for means of 2^6 respectively 2^{31}.[5] For the small mean, the cascades with RLE and DELTA achieve the worst compression rates, while for DICT, FOR and stand-alone NS, the algorithms are roughly grouped by the employed NS algorithm, since DICT and FOR do not change the distributions for the considered mean (see Fig. 3 (h-l)). Regarding the speeds, the top ranks are held by stand-alone NS algorithms. When changing the mean to 2^{31}, the cascades with FOR and DICT achieve by far the best compression rates. Stand-alone NS algorithms are still among the top ranks for the speeds. However, none of them achieves an actual size reduction. While depending on the application, many trade-offs between compression rate and speed could be reasonable, it generally does not make sense, to accept compression rates of more than 32 bits/int, since then, the data could rather be copied or not touched at all, which would be even faster. Keeping this in mind, the cascades with FOR achieve the best results regarding all three speeds, whereby DELTA also makes it into the top-3 for the compression.

Figure 6 shows the results on D5. The cascades of any logical-level technique and SIMD-BP128 achieve better compression rates than the stand-alone SIMD-BP128 from some run length on (Fig. 6 (a)). Regarding the (de)compression speeds, only RLE + SIMD-BP128 can yield an improvement, if the run length exceeds 2^5. It is noteworthy that the cascades with DELTA and FOR imply only a slight slow down, while they achieve much better compression rates. The aggregation speed of RLE + SIMD-BP128 gets out of scope for any other cascade for run lengths above 2^8, since the aggregation of RLE has to execute only one multiplication and one addition *per run*. The next three rows of Fig. 6 compare all cascades for average run lengths of 6, 37, and 517. Even for the lowest of these run lengths (Fig. 6 (e-h)),

[5] The bars in these diagrams are sorted, such that the best algorithm is at the left. We use the color to encode the NS algorithm and the hatch to encode the logical-level technique, whereby *(none)* means a stand-alone NS algorithm. Furthermore, bars with an *X* on top mark algorithms which do not achieve a size reduction on the dataset, i.e., require at least 32 bits per integer.

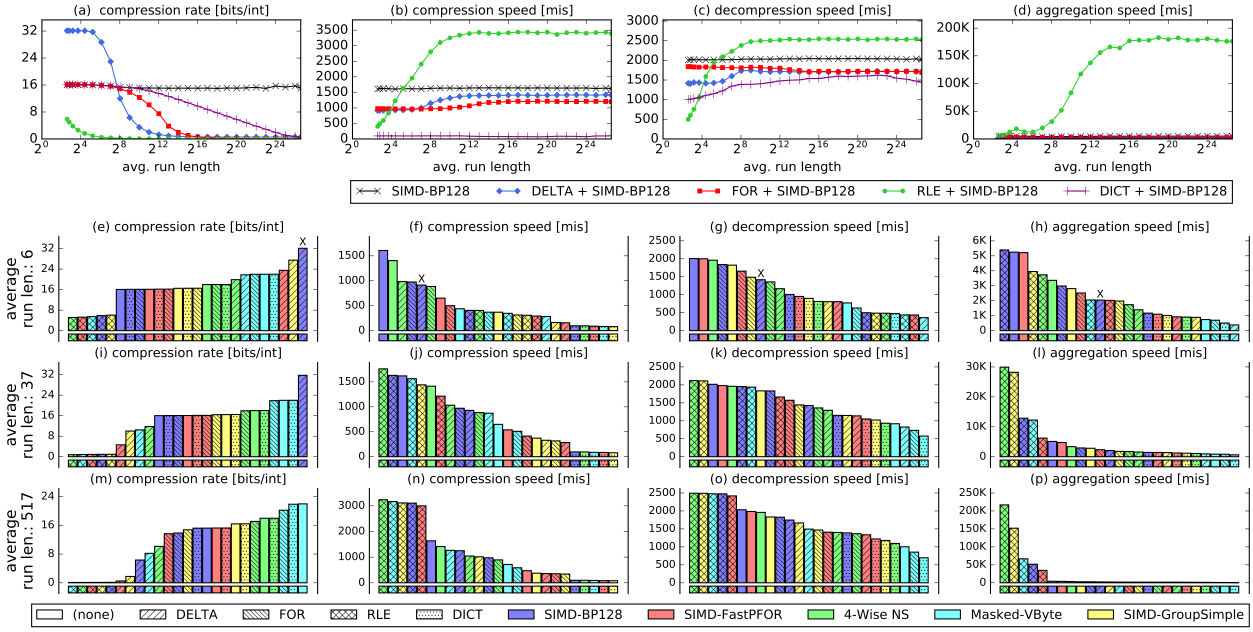


Figure 6: Comparison of the cascades on dataset D5.[5]

the cascades with RLE yield by far the best compression rates, while those with DELTA are among the last ranks. However, the (de)compression speeds of the cascades with RLE are not competitive to those of the best stand-alone NS algorithms. On the other hand, RLE + SIMD-BP128 has the best aggregation speed. As the run lengths get a little higher (Fig. 6 (i-l)), the cascades with RLE move further towards the top-ranks of the speeds and further improve their compression rates. Interestingly, the compression rates of the cascades with DELTA do now achieve the best compression rates after the cascades with RLE, except for DELTA + SIMD-BP128, which still yields the worst compression rate. When the run length is increased further (Fig. 6 (m-p)), these trends continue and the cascades with RLE do now dominate both, the compression rate and all three speeds.

Figure 7 (a-d) report the results of SIMD-BP128 and its cascades on D6 subject to the number of distinct data elements. Since D6 is sorted, a low number of distinct values is equal to a high average run length. Consequently, RLE + SIMD-BP128 achieves a better compression rate than stand-alone SIMD-BP128 until the number of distinct values comes close to the total number of values, i.e. 100 M. Although the possible minimum value is zero, FOR + SIMD-BP128 also improves the compression rate. This is due to the fact that within each input block of the cascade, the value range is small as the data is sorted. Apart from that, especially the decompression speed is interesting. For low numbers of distinct values and thus long runs, SIMD-BP128 and its cascade with RLE are nearly equally fast. As the number of distinct values increases, SIMD-BP128 is affected stronger than RLE + SIMD-BP128. However, when the number of distinct values exceeds 2^{21}, the performance of the cascade with RLE deteriorates and from this point on, the cascade with FOR, respectively DELTA is the fastest algorithm. Note that in this case, the decompression of the stand-alone SIMD-BP128 is never the fastest alternative. Figure 7 (e-h) show the comparison of all 25 algorithms when the dataset contains 128 distinct values. Since the average run length is very high (nearly 800k), the cascades including RLE are the best regarding both, compression rate and speeds. The extreme case of unique data elements, i.e., 100 M distinct values, is given in Fig. 7 (i-l). Now the cascades of RLE are among the worst algorithms for all four measured variables, since the data contains no runs. The best compression rates are now achieved by the cascades of DELTA, since the data is sorted. While the fastest compressor is stand-alone SIMD-BP128, the next ranks are held by cascades making use of DELTA. Regarding the decompression speed, the top-3 algorithms use SIMD-BP128 for the NS-part and DELTA, FOR, or no preprocessing. In terms of the aggregation speed, the stand-alone NS algorithms SIMD-BP128 and SIMD-FastPFOR are the fastest. However, DELTA + SIMD-BP128 and FOR + SIMD-BP128 also achieve very good aggregation speeds, but much better compression rates.

Summing up, the changes to the data distributions achieved by the logical-level techniques do indeed propagate to the compression rates of their cascades with NS. Furthermore, the speeds of the cascades can even exceed those of the corresponding stand-alone NS algorithms. This is especially true for the cascades including RLE, if the data contains long enough runs. Cascades with the other three logical-level techniques generally lead to less significant speed ups or even slow downs, whereby these often come with an improvement of the compression rate. Finally, if the logical-level technique is fixed, its cascades with different NS algorithms can lead to significantly different results regarding compression rate and speed. This justifies the consideration of multiple different NS algorithms even in cascades.

5.4 Lessons Learned

In order to employ lightweight compression effectively, it is desirable to know which algorithm is most suitable for a

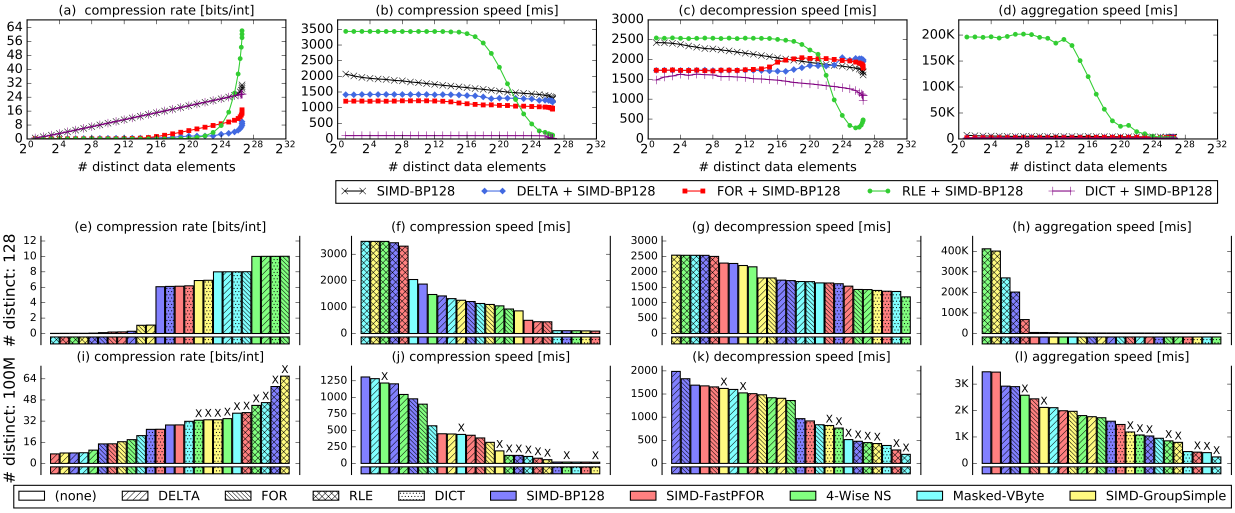


Figure 7: Comparison of the cascades on dataset D6.[5]

given data set w.r.t. a certain optimization goal as, e.g., the best compression rate, the highest (de)compression speed, or a combination thereof. Regarding the compression *techniques*, we can observe some general trends. For instance, NS usually performs the better, the lower the values are, while RLE profits from long runs, and DICT from few distinct values. However, these facts can be derived from the ideas of the techniques and have already been shown experimentally by other authors, e.g. in [1]. What is more interesting is the level of the compression *algorithms*. While SIMD-BP128 seems to be a good choice regardless of the optimization goal *if the data exhibits a good locality*, the case is more complicated for data with a low locality. What makes a decision even more complex is that the performance of some NS algorithms is not monotonic in the size of the values. This holds, e.g., for the word-aligned NS algorithms (Fig. 1 (i-l)) as well as Masked-VByte (Fig. 2, second column).

Moreover, lightweight data compression is still a hot research topic. Hence, more algorithms will be published in the future. Therefore, an automatic approach for choosing the best out of a set of algorithms would be welcome. It is self-evident that the naïve solution of first executing all considered algorithms on the exact data to be compressed and then choosing the best algorithm is infeasible for efficiency reasons. Instead, a model of the algorithms' compression rate and performance – subject to the data properties – could be used. While we believe that such a model could be built based on our systematic evaluation, we see the main contribution of this paper in illustrating that this decision is non-trivial and that, thus, further research in the direction of an automatic selection is necessary. However, since this is beyond the scope of this paper, we leave it for future work.

6. EXPERIMENTS ON REAL DATA

To complement our experiments on synthetic data, we evaluated the algorithms considered in Section 5.3 on a dataset of postings lists of the real-world document collection GOV2[6], which is frequently used to evaluate integer compression algorithms [12, 18, 20]. GOV2 is a corpus of 25 M documents found in a crawl of the `.gov` websites. The dataset contains about 1.1 M postings lists in total, each of which is a *sorted* array of *unique* 32-bit document ids. We discarded all lists containing less than 8192 integers, since time measurements are not reliable enough on too short arrays.

Figure 8 (a-d) compare SIMD-BP128 to its cascaded derivatives subject to the list length. Note that, as the total number of entries in the lists increases, so does the number of distinct entries (uniqueness), while the average difference of two subsequent entries decreases. The compression rate of RLE is noncompetitive to the other algorithms, since unique values imply the absence of runs. Employing any other logical-level technique can yield an improvement of the compression rate, whereby DELTA and FOR get better as the lists get longer. Regarding the compression and aggregation speed, pure SIMD-BP128 is the fastest for all list lengths. However, its cascades with DELTA respectively FOR (only aggregation) are not much slower for long lists. Regarding the decompression, using DELTA or FOR yields better results than pure NS for lists longer than 2^{19} respectively 2^{20}.

Figure 8 (e-h) provide the rankings of all 25 algorithms. The reported measurements are averages over all lists lengths weighted by the actual distribution of the lengths in the dataset. The cascades with DELTA yield the best compression rates. The fastest compressors are SIMD-BP128 and 4-Wise NS followed by their cascades with DELTA. Regarding the decompression and aggregation speeds, the top-4 are stand-alone NS algorithms, which are followed by cascades with DELTA or FOR.

7. CONCLUSION

Lightweight data compression is heavily employed by modern in-memory column-stores in order to compensate for the low main memory bandwidth. In recent years, the corpus of available compression algorithms has significantly grown, mainly due to the use of SIMD extensions. In our experimental survey, we systematically evaluated recent vectorized algorithms of all five basic techniques of lightweight compression as well as cascades thereof on a multitude of synthetic

[6] This data set of postings lists is provided by Lemire et al. at `http://lemire.me/data/integercompression2014.html`.

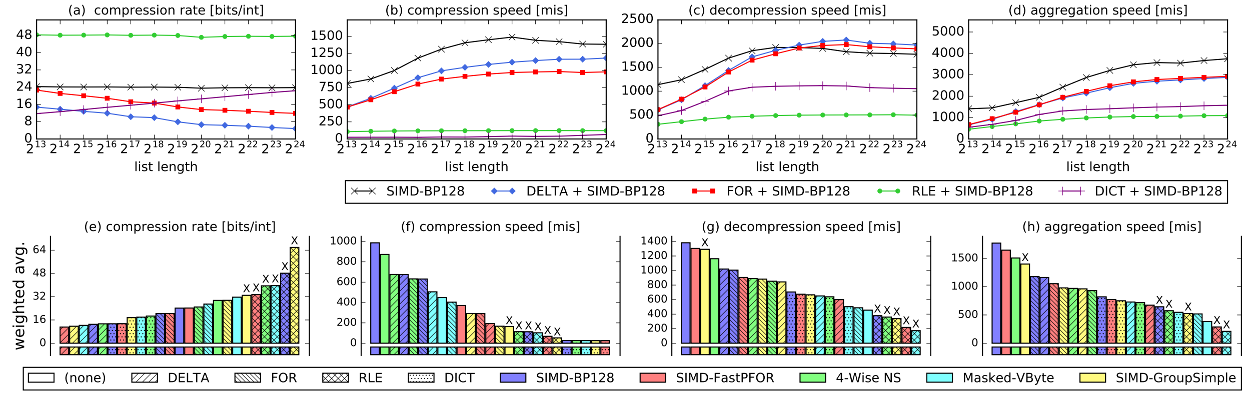


Figure 8: Comparison of the cascades on the postings lists of the real-world document collection GOV2.[5]

and one real data set. We have shown that there is no single-best algorithm suitable for all data sets. Instead, making the right choice is non-trivial and always depends on data properties such as value distributions, run lengths, sorting, and the number of distinct data elements. Furthermore, the best algorithm regarding the compression rate is often not the best regarding the (de)compression speed, such that a trade-off must be defined. Even the various null suppression algorithms show significantly different behavior depending on the data distribution. Finally, cascades of two techniques can heavily improve the compression rate, which comes at the cost of a lower speed in some, but not all cases.

Acknowledgments

This work was partly funded by the German Research Foundation (DFG) in the context of the project ”Lightweight Compression Techniques for the Optimization of Complex Database Queries” (LE-1416/26-1).

8. REFERENCES

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[2] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Softw., Pract. Exper.*, 40(2), 2010.

[3] D. Arroyuelo, S. González, M. Oyarzún, and V. Sepulveda. Document identifier reassignment and run-length-compressed inverted indexes for improved search performance. In *SIGIR*, 2013.

[4] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12), 2008.

[5] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[6] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. *SIGMOD Rec.*, 14(4), 1985.

[7] P. Damme, D. Habich, and W. Lehner. A benchmark framework for data compression techniques. In *TPCTC*, 2015.

[8] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald. Compressed linear algebra for large-scale machine learning. *PVLDB*, 9(12), 2016.

[9] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *ICDE*, 1998.

[10] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9), 1952.

[11] T. Kissinger, T. Kiefer, B. Schlegel, D. Habich, D. Molka, and W. Lehner. ERIS: A numa-aware in-memory storage engine for analytical workloads. In *ADMS*, 2014.

[12] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.*, 45(1), 2015.

[13] D. Lemire, L. Boytsov, O. Kaser, M. Caron, L. Dionne, M. Lemay, E. Kruus, A. Bedini, M. Petri, and R. B. Araujo. The FastPFOR C++ library: Fast integer compression, `https://github.com/lemire/FastPFOR`.

[14] J. Plaisance, N. Kurz, and D. Lemire. Vectorized vbyte decoding. *CoRR*, abs/1503.07387, 2015.

[15] M. A. Roth and S. J. Van Horn. Database compression. *SIGMOD Rec.*, 22(3), 1993.

[16] B. Schlegel, R. Gemulla, and W. Lehner. Fast integer compression using SIMD instructions. In *DaMoN*, 2010.

[17] F. Silvestri and R. Venturini. Vsencoding: Efficient coding and fast decoding of integer lists via dynamic programming. In *CIKM*, 2010.

[18] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. Simd-based decoding of posting lists. In *CIKM*, 2011.

[19] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6), 1987.

[20] W. X. Zhao, X. Zhang, D. Lemire, D. Shan, J. Nie, H. Yan, and J. Wen. A general simd-based approach to accelerating compression algorithms. *ACM Trans. Inf. Syst.*, 33(3), 2015.

[21] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.*, 23(3), 1977.

[22] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.

SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases

Linnea Passing
passing@in.tum.de

Manuel Then
then@in.tum.de

Nina Hubig
hubig@in.tum.de

Harald Lang
langh@in.tum.de

Michael Schreier
schreier@in.tum.de

Stephan Günnemann
guennemann@in.tum.de

Alfons Kemper
kemper@in.tum.de

Thomas Neumann
neumann@in.tum.de

Technical University of Munich

ABSTRACT

Data volume and complexity continue to increase, as does the need for insight into data. Today, data management and data analytics are most often conducted in separate systems: database systems and dedicated analytics systems. This separation leads to time- and resource-consuming data transfer, stale data, and complex IT architectures.

In this paper we show that relational main-memory database systems are capable of executing analytical algorithms in a fully transactional environment while still exceeding performance of state-of-the-art analytical systems rendering the division of data management and data analytics unnecessary. We classify and assess multiple ways of integrating data analytics in database systems. Based on this assessment, we extend SQL with a non-appending iteration construct that provides an important building block for analytical algorithms while retaining the high expressiveness of the original language. Furthermore, we propose the integration of analytics operators directly into the database core, where algorithms can be highly tuned for modern hardware. These operators can be parameterized with our novel user-defined *lambda* expressions. As we integrate lambda expressions into SQL instead of proposing a new proprietary query language, we ensure usability for diverse groups of users. Additionally, we carry out an extensive experimental evaluation of our approaches in HyPer, our full-fledged SQL main-memory database system, and show their superior performance in comparison to dedicated solutions.

CCS Concepts

•**Information systems** → **Online analytical processing engines;** *Main memory engines; Data mining;*

Keywords

HyPer, Computational databases

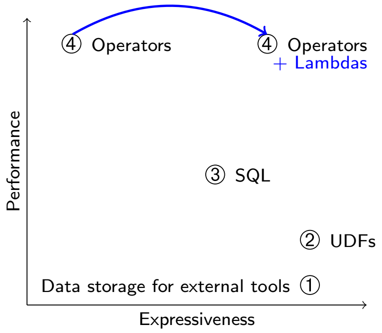


Figure 1: Overview of approaches to data analytics using RDBMS. Our system supports the novel *layer 4*, where data mining is integrated directly into the database core leading to higher performance. To maintain expressiveness, high-order functions (*lambdas*) are passed as parameters.

1. INTRODUCTION

The current data explosion in science and technology poses difficulties for data management and data analytics. Especially stand-alone data analytics applications [2, 16] are prone to have problems due to their simple data management layer. Being optimized for read-mostly or read-only analytics tasks, most stand-alone systems are unsuitable for frequently changing datasets. After each change, the whole data of interest needs to be copied to the application again, a time- and resource-consuming process.

We define *data analytics* to be algorithms and queries that process the whole dataset (or extensive subsets), and therefore are computation-intensive and long-running. This domain contains, for example, machine learning, data mining, graph analytics, and text mining. In addition to the differences between these subdomains, most algorithms boil down to a model-application approach: i.e., a two phase process where a model is created and stored first and then applied to the same or different data in a second step.

In contrast to dedicated analytical systems, classical DBMS provide an efficient and update-friendly data management layer and many more useful features to store big data reliably, such as user rights management and recovery procedures. Database systems avoid data silos as data has to be stored only once, eliminating ETL cycles (extraction, trans-

Series ISSN: 2367-2005 10.5441/002/edbt.2017.09

formation, and loading of data). Thus, we investigate how data analytics can be sensible integrated into RDBMS to contribute to a "one-solution-fits-it-all" system. What level of efficiency is possible when running such complex queries in a database? Can a database actually be better than single purpose standalone systems? According to Aggarwal et al. [4], seamless integration of data analytics technology into DBMS is a major challenge.

Some newer database systems, for example SAP HANA [15] and HyPer [20], are designed to efficiently handle different workloads (OLTP and OLAP) in a single system. Main-memory RDBMS, such as HyPer, are specifically well-suited for high analytics workload due to their efficient use of modern hardware, i.e., multi-core CPUs with extensive instruction sets and large amounts of main memory.

How is an analytics algorithm best integrated into an RDBMS? While existing database systems that feature data analytics include the algorithms on a very high level, we propose to add a specific set of algorithmic building blocks as deep in the system as possible. To describe and assess different approaches of integrating data analytics algorithms into an RDBMS, we distinguish four layers ranging form the least to the most deeply integrated:

(1) DBMS as data storage with external analytics algorithms—the currently most commonly used approach.

(2) User-defined functions (UDFs)—code snippets in high-level languages executed by the DBMS.

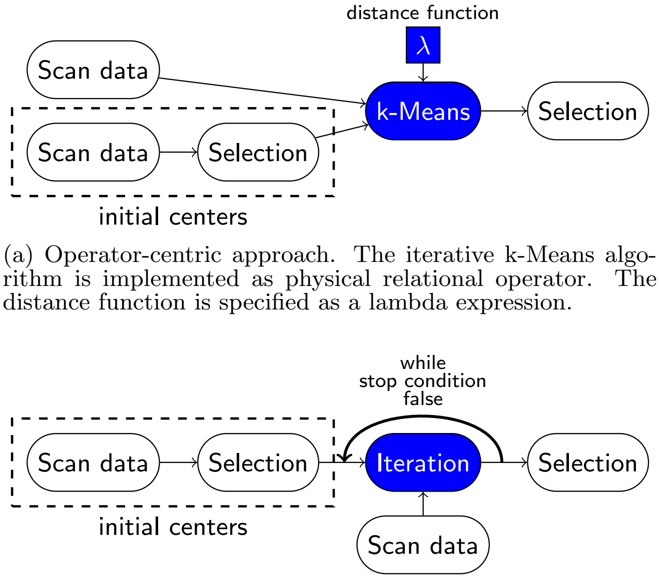
(3) SQL queries—including recursive common table expressions (CTE) and our novel iteration construct.

(4) Integration as physical operators—the deepest integration, providing the highest performance.

We propose user-defined code snippets as parameters to our operators to increase flexibility within *(4)*. These so-called lambda functions, containing for instance distance metrics, are able to change the semantics of a given analytical algorithm.

These four approaches trade performance versus flexibility in a different way, as depicted in Figure 1. We propose implementing several of these approaches into one system to cover the diverse needs regarding performance and expressiveness of different user groups and application domains. The novel operator integration *(4)* combines the highest performance with high flexibility but can only be implemented by the database system's architects. Approaches *(2)* and *(3)* provide environments in which expert users can implement their own algorithms. All three integrated approaches [*(2)*, *(3)*, *(4)*] avoid ETL costs, stale data, and assembling and administrating complex system environments, thereby facilitating ad-hoc data analytics.

This paper focuses on two approaches, SQL- and operator-centric approaches to data analytics in databases. Figure 2 gives a first idea of how these approaches are integrated into query plans. As depicted, both approaches handle arbitrarily pre-processed input. Both approaches result in a relation; this result can thus be post-processed within the same query. The operator-centric approach features a specialized operator that processes data as would any other relational operator such as a join. In the SQL-centric approach, analytical algorithms are expressed in SQL. k-Means is an iterative algorithm, hence Figure 2 shows an iteration as the most important part of the query.



(a) Operator-centric approach. The iterative k-Means algorithm is implemented as physical relational operator. The distance function is specified as a lambda expression.

(b) SQL-centric approach. The iterative algorithm, including initialization and stop condition, is expressed in SQL. The iteration operator can either be the standard recursive common table expression, or our optimized non-appending iteration construct.

Figure 2: Query plans for k-Means clustering.

1.1 Contributions

In this paper, we present how data analytics can efficiently be integrated into relational database systems. Our approach targets different user groups and application domains by providing multiple interfaces for defining and using analytical algorithms. High expressiveness and performance are achieved via a unique combination of existing and new concepts including[1]:

- A classification and assessment of approaches to integrate data analytics with databases.
- The iteration construct as extension to recursive common table expressions (`with recursive`) in SQL.
- Analytical operators executed within the database engine that can be parameterized using lambda expressions (anonymous user-defined functions) in SQL.
- An experimental evaluation with both dedicated analytical systems and database extensions for analytical tasks.

The remainder of this paper is organized as follows: An overview of the related work is provided in Section 2. In Section 3 we discuss what characteristics make HyPer especially suited for in-database analytics. We continue by explaining the in-database processing in Section 4. The method by which analytics are integrated into SQL is defined in Section 5 and our building blocks (operators) of the fourth layer are shown in detail in Section 6. Our operators are very flexible due to their lambda functions, which we describe in Section 7. The evaluation of our operators in HyPer is given in Section 8. We discuss the conclusions of our work in Section 9.

[1] Partially based on our prior publication [29].

2. RELATED WORK

Data analytics software can be categorized into dedicated tools and extensions to DBMS. In this section, we introduce major representatives of both classes.

2.1 Dedicated Data Analytics Tools

The programming languages and environments *R*[2], *SciPy*[3], *theano*[4] and *MATLAB*[5] are known by many data scientists and are readily available. For these reasons, they are heavily used for data analytics, and implementations of new algorithms are often integrated. In addition, these languages and environments provide data visualizations and are well-suited for exploration and interactive analytics. However, their algorithm implementations often are only single-threaded, which is a major drawback concerning currently used multi-core systems and data volumes.

The next group of existing data analytics tools are *data analytics frameworks*. Most representatives of this category are targeted at teaching and research and do not focus on performance for large datasets. Their architecture makes it easy to implement new algorithms and to compare different variants of algorithms regarding quality of results. Notable examples of data analytics frameworks include *ELKI*[6], which supports diverse index structures to speed up analytics, *RapidMiner*[7], used in industry as well as research and teaching, and *KNIME*[8], which allows users to define data flows and reports via a GUI.

Recently, Crotty et al. presented *Tupleware*, a high-performance analytical framework. Tupleware is meant for purely analytical tasks and the system does not take into account transactions.The authors endorse interactive data exploration by not relying on extensive data preparation [12] and by providing a data exploration GUI. Tupleware requires users to annotate their queries with as much semantics as possible: Queries may solely consist of simple building blocks, e.g., `loop` or `filter`, augmented with user-defined code snippets such as comparison functions. Relational operators—the building blocks of SQL queries—are fairly similar to Tupleware's building blocks but are more coarse-grained, more robust against faulty or malicious user input, and can be used in a more general fashion. They therefore do not guarantee as many invariants. SQL implementations of algorithms could be optimized in a similar fashion, although this requires major changes to relational query optimizers.

The cluster computing framework *Apache Spark* [31] supports a variety of data analytics algorithms. Analytical algorithms, contained in the Machine Learning Library (MLlib), benefit from Spark's scale-up and scale-out capabilities. *Oracle PGX*[9] is a graph analytics framework. It can run predefined as well as custom algorithms written in the *Green-Marl* DSL and is focused on a fast, parallel, and in-memory execution. *GraphLab* [22] is a machine learning framework that provides many machine learning building blocks such as regression or clustering, which facilitate building complex applications on top of them. All of these frameworks use dedicated internal data formats making it necessary to use time-consuming data loading steps. Furthermore, the synchronization of results back to the RDBMS is a complex job that often must be implemented explicitly by the user.

2.2 Data Analytics in Databases

In addition to standalone systems there are database systems which contain data analytics extensions. Being faced with the issue of integrating data analytics and relational concepts, the systems mentioned below come up with different solutions: Either analytical algorithms are executed via calls to library functions, or the SQL language is extended with data analytics functions.

MADlib [17] is an example for the second level of our classification, user-defined functions. This library works on top of selected databases and heavily uses data parallel query execution if provided by the underlying database system. MADlib provides analytical algorithms as user-defined functions written in C++ and Python that are called from SQL queries. The underlying database executes those functions but cannot inspect or optimize them. While the output produced by the functions can directly be post-processed using SQL, only base relations are allowed as input to data analytics algorithms. Thus, full integration of the user-defined functions and SQL queries is neither achieved on a query optimization and execution level nor in the language and query layer.

Another example for the UDF category is the *SAP HANA Predictive Analytics Library (PAL)* [25, 15]. This library offers multi-threaded C++ implementations of analytical algorithms to run within the main-memory database system SAP HANA. It is integrated with the relational model in a sense that input parameters, input data, as well as intermediate results and the output are relational tables. The algorithms, so-called *application functions*, are called from SQL code. They are compiled into query plans and executed individually. In contrast to the afore-mentioned MADlib, PAL integrates in one ecosystem only and is therefore capable of connecting to SAP HANA's user and rights management.

Oracle Data Miner [27] is focused on *supervised* machine learning algorithms. Hence, training data is used to create a model that is then applied to test data using SQL functions. Both steps are run multi-threaded to make use of modern multi-core systems. Results of the algorithms are stored in relational tables. Interactive re-using and further processing of results within the same SQL query is not possible, but can be applied in precedent and subsequent queries.

EmptyHeaded [1] uses a datalog-like query language for graph processing. This system follows the "one-solution-fits-all" approach: Graph data is processed in a relational engine using multiway join algorithms that are more suitable for graph patterns than classical pairwise join algorithms.

LogicBlox [6] is also a relational engine that does not use SQL for queries. It relies on functional programming, as does Tupleware, but is a full relational DBMS. The functional programming language of LogicBlox, LogiQL, can be combined with declarative programming and features a relational query optimizer. LogicBlox exploits constraint solving to optimize the functional code.

SimSQL [11] is another recent relational database with analytical features. Users write algorithms from scratch

[2] http://www.r-project.org/

[3] http://www.scipy.org/

[4] http://deeplearning.net/software/theano/

[5] http://www.mathworks.com/products/matlab/

[6] http://elki.dbs.ifi.lmu.de/

[7] http://rapidminer.com/

[8] http://www.knime.org/

[9] http://www.oracle.com/technetwork/oracle-labs/parallel-graph-analytics/

which are then translated into SQL. Several SQL extensions, such as for-each style loops over relations as well as vector and matrix data types, facilitate analytics in the database. While recognizing that its tuple-oriented approach to matrix-based problems results in low performance [10], SimSQL emphasizes its general-purpose approach. As a result of those design decisions, SimSQL is able to execute complex machine learning algorithms which many other computation platforms are not able to do [10], but lacks optimizations for standard analytical algorithms.

To conclude, while all presented database systems strive for integration of analytical and relational queries, the achieved level of integration vastly differs between systems. Most presented systems rely on black box execution of user-defined functions by the database while others transform analytical queries into relational queries to allow for query inspection and optimization by the database.

3. HYPER FOR DATA ANALYTICS

HyPer [20] is a *hybrid* main-memory RDBMS that is optimized for both transactional and analytical workloads. It offers best-in-class performance for both, even when operating simultaneously on the same data. Adding capabilities to execute data analysis algorithms is the next step towards a unified data management platform without stale data.

Several features of HyPer contribute to its suitability for data analytics. First, HyPer *generates efficient data-centric code* from user queries thus reducing the user's responsibility to write algorithms in an efficient way [24]. After transforming the query into an abstract syntax tree (AST), multiple optimization steps, and the final translation into a tree of physical operators, HyPer generates code using the LLVM compiler framework. Computation-intensive algorithms benefit from this design because function calls are omitted. As a result, users without knowledge in efficient algorithms can write fast analytical queries.

Second, *data locality* further improves performance. Data-centric execution attempts keeping data tuples in CPU registers as long as possible to avoid copying of data. If possible, a tuple is kept in registers while multiple operators are executed on it. This so-called *pipelining* is important for queries that touch tuples multiple times. For ad-hoc analytical queries pre- and post-processing steps can be combined with the data processing to generate highly efficient machine code. As many analytical algorithms are pipeline breakers, in practice we pipeline pre-processing and data materialization as well as result generation and post-processing.

Third, HyPer focuses on *scale-up* on multi-core systems rather than on *scale-out* on clusters; hence, parallelization of the operators and the generated code is a performance-critical aspect. Characteristics of modern hardware, such as non-uniform memory access (NUMA), cache hierarchies, and vector processing must be taken into account when new features are integrated into the DBMS. Avoiding data distribution onto multiple nodes is especially important when the input data cannot be chunked easily, e.g., when processing graph-structured data.

In addition to efficient integration of algorithms, other characteristics further encourage the use of HyPer for data analysis use-cases such as: the system provides a PostgreSQL-compatible SQL interface, is fully ACID-compliant and offers fast data loading [23], which is especially important for data scientists.

4. IN-DATABASE PROCESSING

Existing systems for data analysis often use their own proprietary query languages and APIs to specify algorithms (e.g., Apache Spark [31] and Apache Flink [5]). This approach has several drawbacks. For example, unusual query languages make it necessary to extensively train the domain experts that write queries. If common high-level programming languages like Java are used, many programmers are available, but they usually lack domain knowledge. Additionally, optimizing high-level code is a hard problem that compiler designers have been working on for decades, especially in combination with additional query execution logic.

Our goal is to enable data scientists to create and execute queries in a *straightforward* way, while keeping all *flexibility* for expert users. In this chapter, we assess multiple approaches to integrate data analytics into HyPer. The first layer shown in Figure 1 using the database system solely as data storage is omitted here as it does not belong to the in-database processing category. Layers two and three, UDFs and SQL queries, respectively, are already implemented in various database systems. Layer four describes our novel approach of deeply integrating complex algorithms into the database core to maximize query performance while retaining flexibility for the user.

4.1 Program Execution within the Database

Many RDBMS allow user-defined functions (UDFs) in which database users can add arbitrary functionality to the database. This eliminates the need to copy data to external systems. The code snippets are registered with the database system and are usually run by the database system as a black box, although first attempts to "open the black box" have been made [18]. If UDF code contains SQL queries, executing these queries potentially requires costly communication with the database. This is because for most UDF languages it is not possible to bind together the black box code and the code that executes the embedded SQL query thus foregoing massive optimization potential. Because of the dangers to stability and security that go along with executing foreign code in the database core, a sandbox is required to separate database code and user code.

4.2 Extensions to SQL

There is general consensus that relational data should be queried using SQL. By extending SQL to integrate new algorithms, the vast amount of SQL infrastructure (JDBC connectors and SQL editors) can be reused to work with analytical queries. Furthermore, the declarative nature of SQL makes it easy to continuously introduce new optimizations. By using this common language, one avoids the high effort of creating a new language and of teaching it to users.

Some algorithms, such as the a-priori algorithm [8] for frequent itemset mining, work well in SQL but others are difficult to express in SQL and even harder to optimize. One common difficulty is the *iterative nature* of many analytical algorithms. To express iterations in SQL, recursive common table expressions (CTE) can be used. CTEs compute a monotonically growing relation, i.e., tuples of *all* previous iterations are kept. As many iterative algorithms need to access *one* previous iteration only, memory is wasted if the optimizer does not optimize this hard-to-detect situation. This is a problem especially for main-memory databases where memory is a scarce resource.

To solve this issue, we suggest an iteration concept for SQL that does not *append* to the prior iteration but instead *replaces* it and therefore drastically reduces the memory footprint of iterative computations. As the intermediate results become smaller, less data has to be read and processed, thus, non-appending iterations also improve analytics performance. We explain the details of our iteration concept in Section 5.

4.3 Data Analytics in the Database Core

In contrast to other database systems, HyPer integrates important data analytics functionality directly into the core of the database system by implementing special highly-tuned *operators* for analytical algorithms. Because the internal structures of database systems are fairly different, such operators have to be specifically designed and implemented for each system. Differentiating factors between systems are, among others, the execution model (tuple-at-a-time vs. vectorized execution) as well as the storage model (row store vs. column store). For example, an operator in the analytical engine Tupleware, which does not support updating datasets, would look significantly different from an operator in the full-fledged database system HyPer, which needs to take care of updates and query isolation.

HyPer can arbitrarily mix relational and analytical operators leading to a seamless integration between analytics with other SQL statements into one query plan. This is especially useful because the functionality of existing relational operators can be reused for common subtasks in analytical algorithms, such as grouping or sorting. Analytical operators can focus on optimizing the algorithm's core logic such as providing efficient internal data representations, performing algorithm-dependent pre-processing steps, and speeding up computation-intensive loops. A further advantage of custom-built analytical operators is that the query optimizer knows their exact properties and can choose an optimal query plan based on this information. Having all pre- and post-processing steps in one language—and one query—greatly simplifies data analytics and allows efficient ad-hoc queries. In Section 6 we elaborate on our implementation of (physical) operators.

Of the integration layers presented in this section, special operators are integrated most deeply into the database. As a result, they provide unrivaled performance but reduce the user's flexibility. To regain flexibility, we propose lambda expressions as a way of injecting user-defined code into operators. Lambdas can, for example, be used to specify distance metrics in the k-Means algorithm or to define edge weights in PageRank.

By implementing multiple layers, we can offer data analytics functionality to diverse user groups. User-defined algorithms are attractive for data scientists wanting to implement specific algorithms in their favorite programming language without having to copy the data to another system. Persons knowledgeable in analytical algorithms and SQL might prefer to stick to their standard data querying language making extensions to SQL their best choice. Algorithm operators implemented by the database developers are targeted towards users that are familiar with the data domain but not with data analytics algorithm design.

Syntactically, UDFs, stored SQL queries and special operators cannot and should not be distinguished by the user. In this way, DBMS architects can decide on an algorithm's level of integration, which is transparent to the user.

In the following sections, we delve into the details of data analytics using SQL and using specialized analytical operators with λ-expressions. We omit the details on the first two layers—using the database solely as data storage, and running UDFs in a black box within the database—because the first layer does not incorporate any analytical algorithms on the database system side and the second layer uses the database system as a runtime environment for user-defined code. When the database is only used to provide the data, the performance is bound by data transfer performance and the data analytics software used to run the algorithms. In case the database is used to execute code in a black box, again, the runtime depends on the programming language and implementation used in these UDFs.

5. DATA ANALYTICS USING SQL

Our overall goal is to seamlessly integrate analytical algorithms and SQL queries. In the third layer, which is described in this section, SQL is used and extended to achieve this goal. Standard SQL provides most functionality necessary for implementing analytical algorithms, such as fix point recursion, aggregation, sorting, or distinction of cases. However, one vital construct is missing: a more general concept of iteration has to be added to the language. Section 5.1 introduces this general-purpose iteration construct, called *iterate operator*. Query optimization for analytical queries is discussed in Section 5.2.

Our running example is the three algorithms k-Means, Naive Bayes, and PageRank which are well-known [30, 3] examples from vector and graph analytics and used as example building blocks in other state-of-the-art analytics systems [12]. Their properties are shared by many other data mining and graph analytics algorithms. Furthermore, they represent the areas of data mining, machine learning, and graph analytics. Thus, these three algorithms are appropriate examples for this paper.

5.1 The Iterate Operator

The SQL:1999 standard contains recursive common table expressions (CTE) that are constructed using the `with recursive`. Recursive CTEs allow for computation of growing relations, e.g., transitive closures. In these queries, the CTE can be accessed from within its definition and is iteratively computed until no new tuples are created in an iteration. In other words, until a fixpoint is reached. Although it is possible to use this fixpoint recursion concept for general-purpose iterations, this is clearly a diversion from its intended use case, and can thus result in incorrect optimizer decisions.

Our iterate operator has similar capabilities as recursive CTEs: it can reference a relation within its definition allowing for iterative computations. In contrast to recursive CTEs, the iterate operator *replaces* the old intermediate relation rather than *appending* new tuples. Its final result is a table with the tuples that were computed in the last iteration only. This pattern is often used in data and graph mining algorithms, especially when some kind of metric or quality of data tuples is computed. In PageRank, for example, the initial ranks are updated in each iteration. In clustering algorithms, the assignment of data tuples to clusters has to be updated in every iteration. These algorithms have in common that they operate on fixed-size datasets, where

```
SELECT * FROM ITERATE([initialization], [step], [stop
      condition]);

-- find smallest three-digit multiple of seven
SELECT * FROM ITERATE((SELECT 7 "x"),
      (SELECT x+7 FROM iterate),
      (SELECT x FROM iterate WHERE x>=100));
```

Listing 1: Syntax of the ITERATE SQL language extension. A temporary table *iterate* is created, that in the beginning contains the result of the *initialization* subquery. Iteratively, the subquery *step* is applied to the result of the last iteration, until the boolean condition *stop condition* is fulfilled.

only certain values (ranks, assigned clusters, et cetera) are updated. This means the stop criterion has to be changed; rather than stopping when no new tuples are generated, our iterate operator stops when a user-defined predicate evaluates to true. We show the syntax of the iteration construct in Listing 1. By providing a non-appending iteration concept with a while-loop-style stop criterion, we are adding more semantics to the implementation, which has been shown to massively speed up query execution due to better optimizer decisions [12].

Although it is possible to implement the afore-mentioned algorithms using recursive CTEs, the iterate operator has two major advantages:

- Lower memory consumption: Given a dataset with n tuples, and i iterations. With recursive CTEs, the table is growing to $n * i$ tuples. Using our operator, the size of the relation remains n. For comparisons of the current and the last iteration, we need to store $2*n$ tuples and discard all prior iterations early. The iterate construct saves vast amounts of memory in comparison to recursive CTEs. Furthermore, if the stop criterion is the number of executed iterations, recursive CTEs have to carry along an iteration counter, which is a huge memory overhead because it has to be stored in every tuple.

- Lower query response times: Because of the smaller relation size, our algorithm is faster in scanning and processing the whole relation, which is necessary to re-compute the ranks, clusters, et cetera.

Lower memory requirements are particularly important in main-memory databases like HyPer, where memory is a scarce resource. This is especially true when whole algorithms are integrated into the database because they often need additional temporary data structures. Our evaluation, Section 8, shows how algorithm performance can be improved by using our iterate operator instead of recursive CTEs, while keeping the flexibility of `with recursive` statements. Both approaches share one drawback, they can both produce infinite loops. Those situations need to be detected and aborted by the database system, e.g., via counting recursion depth or iterations, respectively.

A conceptually similar idea that also features appending and non-appending iterations can be found in the work of Binnig et al. [7]. Being a language proposal for a functional extension to SQL, their paper neither discusses where which type of iteration is appropriate, nor does it list advantages and drawbacks regarding performance or memory consumption. Ewen et al. [14] also argue that many algorithms only need small parts of the data to compute the next iteration (so-called *incremental iterations*). Their work focuses on parallelizing those iterations as they are only sparsely dependent on each other. The *SciDB* engine features support for iteration processing on arrays where "update operations typically operate on neighborhoods of cells" [26]. Soroush et al.'s work enables efficient processing of this type of array iterations as well as incremental iterations.

5.2 Query Optimization and Seamless Integration with the Surrounding SQL Query

Keeping intermediate results small by performing selections as early as possible is a basic principle of query optimization. This technique, called pushing selections, is in general not possible when analytical algorithms are affected. This is because the result of an analytical algorithm is not determined by *single tuples* (as it is for example for joins), but potentially influenced by the *whole* input dataset. A similar behavior can be found in the group-by operator, where the aggregated results also depend on *all* input tuples. This naturally narrows the search space of the query optimizer and reduces optimization potentials.

One major influencing factor for query optimization is the cardinality of intermediate results. For instance, precise cardinality estimations are necessary for choosing the best join ordering in a query. It is, however, hard to estimate the output cardinality of the generic iterate operator because it can contain diverse algorithms. Some algorithms, e.g., k-Means, iterate over a given dataset and the number of tuples stays the same before and after the iterate operator. Other algorithms, e.g., reachability computations, increase the dataset with each iteration, which makes the final cardinality difficult to estimate. Cardinality estimation on recursive CTEs faces the same difficulty so that similar estimation techniques can be applied.

To conclude, the diverse nature of analytical algorithms does not offer many generic optimization opportunities. Instead, relational query optimization has to be performed almost independently on the subqueries *below* and *above* the analytical algorithm while the analytical algorithm itself might benefit from different optimization techniques, e.g., borrowed from general compiler design or constrained solving as suggested by [6]. Because of the lacking potential for standard query optimization, low-level optimizations such as vectorization and data locality, as introduced in Section 3, become more important.

6. OPERATORS

The most in-depth integration of analytical algorithms into a DBMS is by providing implementations in the form of physical operators. Physical operators like hash join or merge sort are highly optimized code fragments that are plugged together to compute the result of a query. All physical operators, including the analytical ones introduced in this paper, use tables as input and output. They can be freely combined ensuring maximal efficiency. Figure 3 shows how physical analytics operators are integrated into query translation and execution. Physical operators are performance-wise superior to the general iteration construct, introduced in Section 5.1, as these specialized operators know

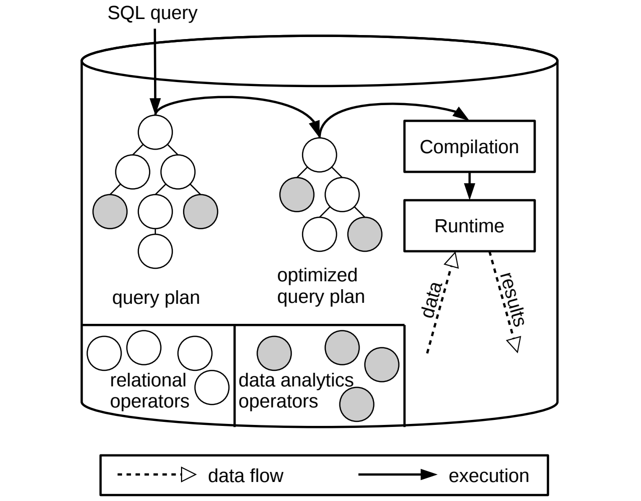


Figure 3: Query translation and execution with relational and analytical operators. A SQL query is translated to an abstract syntax tree (AST) consisting of both relational and analytical operators. The optimizer can inspect both types of operators. This approach provides highest integration and performance.

```
SELECT * FROM PAGERANK((SELECT src, dest FROM edges),
      0.85, 0.0001);
```

Listing 2: Operator integration in SQL. Arbitrary preprocessing of input data and arbitrary postprocessing of the results is possible. Additional parameters define the algorithm's behavior.

invariants of their algorithms such as the estimated output cardinality or data dependencies in complex computations. These specialized operators know best how to distribute work among threads or how to optimize the memory layout of internal data structures.

For example, the query shown in Listing 2 computes the PageRank value for every vertex of the graph induced by the *edges* relation[10]. The query is processed by a table scan operator followed by our specialized physical PageRank operator. The PageRank operator implementation defines, for example, whether parallel input (from the table scan operator) can be processed, information that is used by the optimizer to create the best plan for the given query.

In the next sections, we describe the chosen algorithms, k-Means, Naive Bayes, and PageRank, and how we implemented them in HyPer. Furthermore, we describe necessary changes to the optimizer.

6.1 The Physical k-Means Operator

k-Means is a fast, model-based iterative clustering algorithm, i.e., it divides a set of tuples into k spherical groups such that the sum of distances is minimized. It can be utilized as a building block for advanced clustering techniques. The classical k-Means algorithm by Lloyd [21] splits each iteration into two steps: assignment and update. In the assignment step, each tuple is assigned to the nearest cluster center. In the update step, the cluster centers are set to be the arithmetic mean of all tuples assigned to the cluster. The algorithm converges when no tuple changes its assigned cluster during an iteration. For practical use, the convergence criterion is often softened: Either, a maximum number of iterations is given, or the algorithm is interrupted if only a small fraction of tuples changed its assigned cluster in an iteration.

In our implementation, the k-Means operator requires two input relations, data and initial centers, that are passed via subqueries. An additional parameter defines the maximum number of iterations. Using parallelism, our implementation benefits from modern multi-core systems. Each thread locally assigns data tuples to their nearest center and to prepare the re-computation of cluster centers, each thread sums up the tuples values. The data tuples themselves are consumed and directly thrown away after processing. For the next iteration, tuples are requested again from the underlying subquery. As a result, the query optimizer can decide to compute the data relation each time, or use materialized intermediate results, whatever is faster in the given query. Data locality is ensured because all centers and intermediate data structures are copied for each thread. Thread synchronization is only needed for the very last steps, global aggregation of the local intermediate results and the final update of the cluster centers. This procedure is repeated until the solution remains stable (i.e., no tuple changed its assignment during an iteration), or until the maximum number of iterations is reached. The operator then outputs the cluster centers.

6.2 The Physical Naive Bayes Operator

Naive Bayes classification aims at classifying entities, i.e., assigning categorical labels to them. Other than k-Means or PageRank, it is a supervised algorithm and consists of two steps performed on two different datasets: First, a dataset A with known labels is used to build a model based on the Bayesian probability density function. Second, the model is applied to a related but un-labeled (thus unknown) dataset B to predict its labels. When implemented in a relational database, one challenge is storing the model as it does not match any of the relational entities, relation or index, completely.

We implemented model creation and application as two separate operators, Naive Bayes training and Naive Bayes testing, respectively. The generation of additional statistical measures is handled by two additional operators that are not limited to Naive Bayes but can be used as a building block for multiple algorithms, for example k-Means.

Similar to k-Means, the Naive Bayes training operator is a pipeline breaker. Each thread holds a hash table to manage its input data with the class as key while not storing the tuples itself. In addition, the number of tuples N is stored for each class, as well as the sum of the attribute values $\sum_{n\in N} n.a$ and the sum of the square of each attribute value $\sum_{n\in N} n.a^2$ for each class and attribute. After the whole input is consumed, the training operator computes the a-priori probability for each class as well as the mean and standard deviation for each class and attribute:

[10] Parentheses around the subquery are necessary because arbitrary queries are allowed there. The sole use of commas would have lead to an ambiguous grammar.

Let a given training set D with $|D|$ instances $d \in D$ contain a set of classes C with $|C|$ instances $c \in C$. Let $|c|$ denote the number of instances of this class c in D. Then, the a-priori probability of class c is given by:

$$PR(c) = \frac{|c| + 1}{|D| + |C|}$$

Afterwards, the results and the class labels are fed into the next operator: the testing operator.

6.3 The Physical PageRank Operator

PageRank [9] is a well-known iterative ranking algorithm for graph-structured data. Each vertex v in the graph (e.g., a website or a person), is assigned a ranking value that can be interpreted as its importance. The rank of v depends on the number and rank of incoming edges, i.e., v is important if many important vertices have edges to it. A PageRank iteration is a sparse matrix-vector multiplication. In each iteration, part of each vertex's importance flows off to the vertices it is adjacent to, and in turn each vertex receives importance from its neighbors. Similar to k-Means, PageRank converges towards a fixpoint, i.e., the vertex ranks change less than a user-defined epsilon. It is common to specify a maximum number of iterations.

The sparse matrix-vector multiplication performed in the PageRank iterations is similar to many graph algorithms in that its performance greatly benefits from efficient neighbor traversals. This means for a given vertex v it has to be efficiently possible to enumerate all of its neighbors. Our PageRank implementation ensures this by efficiently creating a temporary compressed sparse row (CSR) representation [28] that is optimized for the query at hand. We avoid storage overhead and an access indirection in this mapping by re-labeling all vertices and doing a direct mapping. After the PageRank computation we use a reverse mapping operator that translates our internal vertex ids back to the original ids.

The PageRank operator uses only the CSR graph index and no longer needs to access the base data. In each iteration we compute the vertices' new PageRank values in parallel without any synchronization. Because we have dense internal vertex ids we are able to store the current and last iteration's rank in arrays that can be directly indexed. Thus, every neighbor rank access only involves a single read. At the end of each iteration we aggregate each worker's data to determine how much the new ranks differ from the previous iterations. If the difference is less or equal to the user-defined epsilon or if the maximum iteration count is reached, the PageRank computation finishes.

7. LAMBDA EXPRESSIONS

In Section 4.3 we described the integration of specialized data analytics operators into the database core. These operators provide unrivaled performance in executing the algorithms they were designed for. However, without modification they are not flexible, i.e., they are not even applicable in the context of similar but slightly different algorithms. Consider the k-Medians algorithm. It is a variant of k-Means that uses the L1-norm (Manhattan distance) rather than the L2-norm (Euclidean distance) as distance metric. While this distance metric differs between the variants, their implementations have in common predominant parts of their code. Even though this common code could be shared, different metrics would make necessary different variants of our algorithm operators.

```
CREATE TABLE data (x FLOAT, y INTEGER, z FLOAT,
                   desc VARCHAR(500));
CREATE TABLE center (x FLOAT, y INTEGER, z FLOAT);
INSERT INTO data ...
INSERT INTO center ...

SELECT * FROM KMEANS(
  -- sub-queries project the attributes of interest
  (SELECT x,y FROM data),
  (SELECT x,y FROM center),
  -- the distance function is specified as λ-expression
  λ(a,b) (a.x-b.x)^2+(a.y-b.y)^2,
  -- termination criterion: max. number of iterations
  3
);
```

Listing 3: Customization of the k-Means operator using a lambda expression for the distance function.

Instead, when designing data analytics operators, we identify and aim to exploit such similarities. Our goal is to have one operator for a whole class of algorithms with variation points that can be specified by the user. To inject user-defined code into variation points of analytics operators we propose using *lambda expressions* in SQL queries.

Lambda expressions are anonymous SQL functions that can be specified inside the query. For syntactic convenience, the lambda expressions' input and output data types are automatically inferred by the database system. Also, for all variation points we provide default lambdas that are used should none be specified. Thus, non-expert users can easily fall back to basic algorithms. With lambda-enabled operators we strive not only to keep implementation and maintenance costs low, but especially to offer a wide variety of algorithm variants required by data scientists. Also, because lambda functions are specified in SQL, they benefit from existing relational optimizations.

Listing 3 shows how our k-Means operator benefits from lambdas. In the `kmeans` function call's third argument, a lambda expression is used to specify an arbitrary distance metric. The operator expects a lambda function that takes two tuple variables as input arguments and returns a (scalar) float value. At runtime, these variables are bound with the corresponding input tuples to compute the distance. Thus, by providing a k-Means operator that accepts lambda expressions we do not only cover the common k-Means and k-Medians algorithms but also allow users to design algorithms that are specific to their task and data at hand. These custom algorithms are still executed by our highly-tuned in-database operator implementation and because all code is compiled together, no virtual function calls are involved.

8. EXPERIMENTAL EVALUATION

In this section, we evaluate our implementations of k-Means, PageRank, and Naive Bayes. As introduced in Section 4, we implemented multiple versions of the algorithms, that reflect different depths of integration. We compare our solutions to other systems commonly used by data scientists. This includes middle-ware tools based on RDBMS, analytics software for distributed systems, and standalone data analysis tools.

	#tuples n	#dimensions d	k
Varying number of tuples	160 000	10	5
	800 000	10	5
	4 000 000	10	$5^{\star}$
	20 000 000	10	5
	100 000 000	10	5
	500 000 000	10	5
Varying number of dimensions	4 000 000	3	5
	4 000 000	5	5
	4 000 000	10	$5^{\star}$
	4 000 000	25	5
	4 000 000	50	5
Varying number of clusters	4 000 000	10	3
	4 000 000	10	$5^{\star}$
	4 000 000	10	10
	4 000 000	10	25
	4 000 000	10	50

$\star$ same experiments, for connecting the three lines of experiments

Table 1: Datasets for k-Means experiments.

8.1 Datasets and Parameters

We use a variety of datasets to evaluate the influence of certain characteristics of the datasets and workload to the resulting performance.

8.1.1 k-Means Datasets and Parameters

k-Means is an algorithm targeted at *vector data*, i.e., tuples with a number of dimensions. This data model fits perfectly into relations. The data is characterized by the number of tuples n, the number of dimensions d used for clustering, and the data types of the dimensions. We chose to perform experiments for varied n and d while keeping the data types constant. In addition to the dataset, the algorithm itself has multiple parameters: the number of clusters k, the cluster initialization strategy, and the number of iterations i that are computed. The number of clusters k drastically influences the query performance because it defines the number of distances to be computed and compared, and is an important parameter in our evaluation. To produce comparable results with a wide range of systems, our experiments use the simplest cluster initialization strategy: random selection of k initial cluster centers. We chose to perform three iterations i, which keeps the experiment duration short while leveling out a possible overhead in the first iteration.

While modifying one parameter, we keep the other two fixed to focus on the effect on that parameter only. The resulting list of experiments is shown in Table 1. We conduct five to six experiments per parameters, which allows us to assess not only the performance but also the scaling behavior of the different systems. The dataset sizes, determined by n and d, were chosen to be processable by all evaluated systems within main memory and within a reasonable time given the vast performance differences between the systems. We create artificial, uniformly distributed, datasets because they provide an important advantage over real-world datasets in our use case. As the performance of plain k-Means with a fixed number of iterations is irrespective of data skew, our decision to use synthetic datasets does not introduce any drawbacks.

8.1.2 Naive Bayes Datasets and Parameters

The Naive Bayes experiments are conducted using the same synthetic datasets as k-Means. We vary the number of tuples N and the number of dimensions d. For the labels we chose a uniform probability density function of two labels 0 and 1. Our experiments cover the *training* phase of the algorithm only as it has a much higher complexity and thus runtime than the *testing* step.

8.1.3 PageRank Datasets and Parameters

PageRank is an algorithm targeted at *graph data*, i.e., vertices and edges with optional properties. The algorithm is parameterized with the damping factor d modeling the probability that an edge is traversed, e, the maximum change between two iterations for which the computation continues, and the maximum number of iterations i. For the damping factor d we chose the reasonable value 0.85 [9], i.e., the modeled random surfer continues browsing with a probability of 85%. To better compare different systems, we set e to 0 and run a fixed number of 45 iterations in all systems. As datasets we use the artificial LDBC graph designed to follow the properties of real-world social networks. We generated multiple LDBC graphs in different sizes up to 500,000 vertices and 46 million edges, using the SNB data generator [13], and used the resulting undirected person-knows-person graph.

8.2 Evaluated Systems

We evaluate our physical operators, denoted as *HyPer Operator*, SQL queries with our iterate operator, denoted as *HyPer Iterate*, and a pure SQL implementation using recursive CTEs, denoted as *HyPer SQL*, against diverse data analysis systems introduced in Section 2. We chose *MATLAB R2015* as a representative of the "programming languages" group. The next category is "big data analytics" platforms, in which we evaluate *Apache Spark 1.5.0* with MLlib. As contender in the "database extensions" category, we chose *MADlib 1.8* on top of the Pivotal Greenplum Database 4.3.7.1.

To ensure a fair comparison, all systems have to implement the same variant of k-Means: Lloyd's algorithm. Note that we therefore disabled the following optimizations implemented in Apache Spark MLlib. First, the MLlib implementation computes lower bounds for distances using norms reducing the number of distance computations. Second, distance computation uses previously computed norms instead of computing the Euclidean distance (if the error introduced by this method is not too big). *litekmeans*[11], a fast k-Means implementation for MATLAB, uses the same optimizations. We therefore use MATLAB's built-in k-Means implementation in our experiments.

8.3 Evaluation Machine

All experiments are carried out on a 4-socket Intel Xeon E7-4870 v2 (15×2.3 GHz per socket) server with 1 TB main memory, running Ubuntu Linux 15.10 using kernel version 4.2. Greenplum, the database used for MADlib, is only available for Red Hat-based operating systems. We therefore set

[11] http://www.cad.zju.edu.cn/home/dengcai/Data/Clustering.html

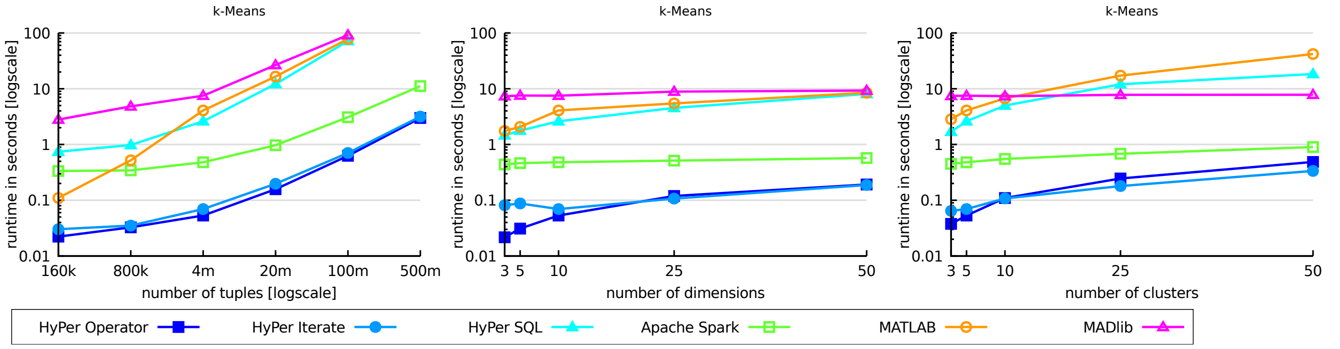


Figure 4: k-Means experiments. From left to right: varying the number of tuples N, dimensions d, and clusters k. Default parameters: 4,000,000 tuples, 10 dimensions, 5 clusters, 3 iterations.

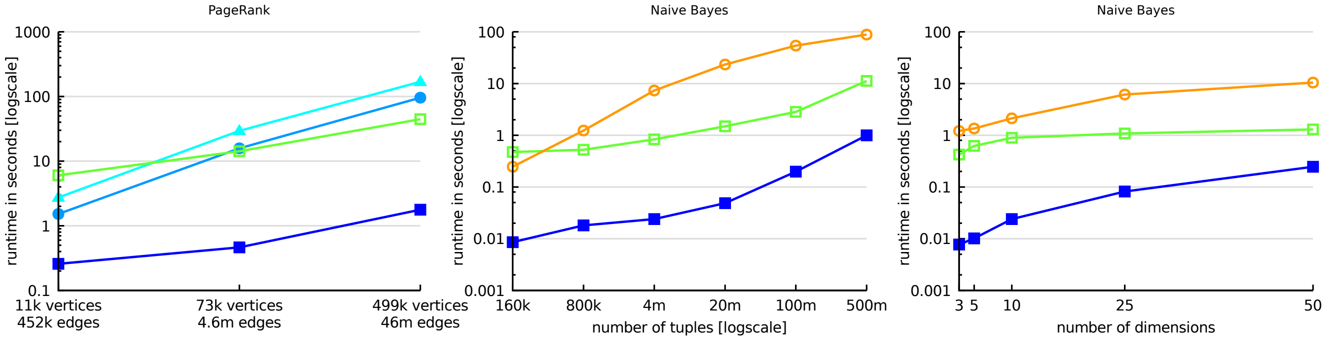


Figure 5: PageRank and Naive Bayes experiments. From left to right: PageRank using the LDBC SNB dataset, damping factor 0.85, and 45 iterations. Naive Bayes experiment varying the number of tuples N. Naive Bayes experiment varying the number of dimensions d.

up a Docker container running CentOS 7. The potential introduced overhead is considered in our discussion. As mentioned, we chose the datasets to fit into main memory, even when considering additional data structures. MATLAB does not contain parallel versions of the chosen algorithms, as mentioned in Section 2. This issue is also considered in the discussion of our results.

8.4 Results and Discussion

Figures 4 and 5 display the total measured runtimes. In general, the results match our claims regarding the four layers of integration as shown in Figure 1: Systems using UDFs (layer 2), in our experiments represented by MADlib, are slower than HyPer Iterate and HyPer SQL using SQL (layer 3). The fastest implementation, HyPer Operator, uses analytical operators (layer 4). Runtime of dedicated analytical systems, such as MATLAB and Apache Spark, heavily depends on the individual system.

8.4.1 Recursive CTEs and HyPer Iterate

As claimed in Section 5, using the iteration concept improves runtimes over plain SQL. While the pure SQL implementation, using recursive CTEs, has to store and process intermediate results that grow with each iteration, the iteration operator's intermediate results have constant size. In our implementations this means additional selection predicates for the pure SQL variant and more expensive aggregates due to the larger intermediate results. k-Means is more affected by this difference because it operates on larger data and is less computation-intensive than PageRank.

8.4.2 Hyper Operator and HyPer Iterate

The k-Means experiments show almost no difference between the HyPer Operator and the HyPer Iterate approach. k-Means is a rather simple algorithm: there is no random data access, only few branches, vectorization can be applied easily, and the data structures are straightforward. Furthermore, k-Means operates on vector data; both operator and SQL implementations use similar internal data structures. This results in very similar code being generated by the operator and the query optimizer resulting in the similar runtimes.

For PageRank, the experiments reveal a different picture: HyPer Operator runs significantly faster than HyPer Iterate because of its optimized CSR graph data structure. In contrast, HyPer Iterate has to work on relational structures, an edges table and a derived vertices table, and subsequently needs to perform many (hash) joins. As a result, its runtime is dominated by building and probing hash tables. This behavior is also found in [19] where a SQL implementation of PageRank also showed performance only comparable to stand-alone-systems. The following rule of thumb can be applied: The more similar optimized SQL code and code generated from the hand-written operator are, the smaller the runtime difference between HyPer Iterate and HyPer Operator approaches.

8.4.3 HyPer, MATLAB, MADlib, and Apache Spark

Among the contender systems, Apache Spark shows by far the best runtimes, which was expected because Spark was especially built for these kinds of algorithms. Still, Apache Spark is multiple times slower than our HyPer Operator approach for all three evaluated algorithms, as shown in Figures 4 and 5. HyPer's one-system-fits-all approach comes with some overhead of database-specific features not present in dedicated analytical systems like Apache Spark. Therefore, it is important that these features do not cause overhead when they are not used. For instance, isolation of parallel transactions should not take a significant amount of time when only one analytical query is running. Some database-specific overhead, stemming for example from memory management and user rights management, cannot be avoided. Nevertheless, HyPer shows far better runtimes than dedicated systems, while also avoiding data copying and stale data. MATLAB runs both algorithms single-threaded and therefore cannot compete, but was included because multiple heavily used data analytics tools do not support parallelism. MADlib, even taking into account the runtime impairment caused by the virtualization overhead, cannot compete with solutions that integrate data analytics deeper and produce better execution code.

Interestingly, Spark and MADlib almost seem not to be affected by the number of dimensions or clusters in the experiments. As algorithm-wise more complex computations are necessary if either of the numbers increases, we suspect those computations to be hidden behind multi-threading overhead. For example, if each thread handles one cluster, even the 50 clusters in the largest experiment still fit into the 120 hyper-threads of the evaluation machine. But k-Means with larger number of dimensions or clusters is not common, because their results are impaired by the curse of dimensionality or cannot be interpreted by humans. Regarding the scaling for larger datasets, log-scaled runtimes fail to show runtime differences appropriately. Plots with log-scaled runtimes counter-intuitively show converging lines when in fact the runtime difference between two systems is constant, which is the case for HyPer Operator/Iterate and Apache Spark in the leftmost sub-figure of Figure 4.

The results presented above support our claim that a multi-layer approach helps targeting diverse user groups. DBMS manufacturers benefit from the identical interface and syntax of UDFs, stored SQL queries, and hard-coded operators. The decision as to in which layer an algorithm should be implemented is solely affected by the implementation effort versus the gain in performance and flexibility. Laypersons can use these manufacturer-provided algorithms without having to care whether it is a UDF, an SQL query, or a physical operator. Database users with expertise, opposed to laypersons wanting to implement their own analytical algorithms can choose to implement either UDFs or SQL queries.

Briefly stated, the experiments match the expected order of runtimes: the deeper the integration of data analytics, the faster the system. Our results also support our idea of one database system being sufficient for multiple workloads. While this has been shown for combining OLTP and OLAP workloads [15, 20], our contribution was to integrate one more workload, data analytics, while keeping performance and usability on a high level.

9. CONCLUSION

We described multiple approaches of integrating data analytics into our main-memory RDBMS HyPer. Like most database systems, HyPer can be used as a data store for external tools. However, doing so exposes data transfer as a bottleneck and prevents significant query optimizations. Instead, we presented three layers of integrating data analytics directly into the database system: data analytics *in UDFs*, data analytics *in SQL*, and analytical *operators in the database core*. The layers' depth of integration and their analytics performance increases with each layer.

UDFs allow the user to implement arbitrary computations directly in the database. However, because the database runs UDFs as a black box, automated optimization potentials are limited. To prevent this lack of optimization potential, we proposed performing data analytics in SQL. As iterations are hard to express in SQL and difficult to optimize, we presented the *iteration operator* and a corresponding language extension that serves as a building block for arbitrary iterative algorithms directly in SQL. Compared to recursive common table expressions, the iteration construct significantly reduces runtime overhead, especially in terms of memory consumption, as it only materializes the intermediate results of the *previous* iteration.

For major analytical algorithms that are used frequently (e.g., k-Means, PageRank, and Naive Bayes), we proposed an even deeper integration: integrating highly-tuned analytical operators into the database core. Using our novel SQL *lambda expressions*, users can specialize analytical operators directly within their SQL queries. This adds flexibility to otherwise fixed operators and allows, for example, for applying arbitrary user-defined distance metrics in our tuned k-Means operator. Just like the iterate operator and the analytics operator, lambda expressions are part of the logical query plan and are subject to query optimization and code generation. Hence, they benefit from decades of research in database systems.

Our presented approaches enable complete integration of data analytics in SQL queries, ensuring both efficient query plans and usability. In our experiments we saw that HyPer data analytics on both graph and vector data is significantly faster than in dedicated state-of-the-art data analytics systems: 92 times faster than Apache Spark for PageRank. This is especially significant because as an ACID-compliant database, HyPer must also be able to handle concurrent transactional workloads. Thus, we showed that HyPer is suitable for integrated data management *and* data analytics on large data, with multiple interfaces targeted at different user groups.

Acknowledgment

This research was supported by the German Research Foundation (DFG), Emmy Noether grant GU 1409/2-1, and by the Technical University of Munich - Institute for Advanced Study, funded by the German Excellence Initiative and the European Union Seventh Framework Programme under grant agreement number 291763, co-funded by the European Union. Further, it was supported by the German Research Foundation (DFG), grant NE 1677/1-1. Manuel Then is a recipient of the Oracle External Research Fellowship. Linnea Passing has been sponsored in part by the German Federal Ministry of Education and Research (BMBF), grant TUM: 01IS12057.

10. REFERENCES

[1] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. EmptyHeaded: A Relational Engine for Graph Processing. In *Proc. SIGMOD 2016*, pages 431–446, 2016.

[2] E. Achtert, H. Kriegel, and A. Zimek. ELKI: A Software System for Evaluation of Subspace Clustering Algorithms. In *Proc. SSDBM 2008*, volume 5069 of *LNCS*, pages 580–585, 2008.

[3] C. C. Aggarwal and H. Wang. Graph Data Management and Mining: A Survey of Algorithms and Applications. In *Managing and Mining Graph Data*, Advances in Database Systems, pages 13–68. 2010.

[4] N. Aggarwal, A. Kumar, H. Khatter, and V. Aggarwal. Analysis the effect of data mining techniques on database. *Advances in Engineering Software*, 47(1):164–169, 2012.

[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. *VLDBJ*, pages 939–964, 2014.

[6] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the logicblox system. In *Proc. SIGMOD 2015*, pages 1371–1382. ACM, 2015.

[7] C. Binnig, R. Rehrmann, F. Faerber, and R. Riewe. FunSQL: It is time to make SQL functional. In *Proc. DanaC 2012*, pages 41–46. ACM, 2012.

[8] F. Bodon. A fast apriori implementation. In *Proc. IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI 2003)*, 2003.

[9] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Proc. WWW 1998*, pages 3825–3833, 1998.

[10] Z. Cai, Z. J. Gao, S. Luo, L. L. Perez, Z. Vagena, and C. Jermaine. A Comparison of Platforms for Implementing and Running Very Large Scale Machine Learning Algorithms. In *Proc. SIGMOD 2014*, pages 1371–1382, 2014.

[11] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine. Simulation of database-valued Markov chains using SimSQL. In *Proc. SIGMOD 2013*, pages 637–648. ACM, 2013.

[12] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. The Case for Interactive Data Exploration Accelerators (IDEAs). In *Proc. HILDA 2016*, pages 11:1–11:6, 2016.

[13] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC Social Network Benchmark: Interactive Workload. In *Proc. SIGMOD 2015*, pages 619–630, 2015.

[14] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning Fast Iterative Data Flows. *Proc. VLDB Endow.*, 5(11):1268–1279, 2012.

[15] 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.

[16] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.

[17] 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. *Proc. VLDB Endow.*, 5(12):1700–1711, 2012.

[18] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the Black Boxes in Data Flow Optimization. *Proc. VLDB Endow.*, 5(11):1256–1267, 2012.

[19] A. Jindal, S. Madden, M. Castellanos, and M. Hsu. Graph analytics using vertica relational database. In *Proc. Big Data 2015*, pages 1191–1200, 2015.

[20] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE 2011*, pages 195–206, 2011.

[21] S. Lloyd. Least squares quantization in PCM. *IEEE Trans. Inf. Th.*, 28(2):129–137, 1982.

[22] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Framework For Parallel Machine Learning. *CoRR*, 2014.

[23] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant Loading for Main Memory Databases. *Proc. VLDB Endow.*, 6(14):1702–1713, 2013.

[24] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.*, 4(9):539–550, 2011.

[25] SAP SE. *SAP HANA Predictive Analysis Library (PAL) Reference, SAP HANA Platform SPS 09*, 2014.

[26] E. Soroush, M. Balazinska, S. Krughoff, and A. Connolly. Efficient Iterative Processing in the SciDB Parallel Array Engine. In *Proc. SSDBM 2015*, pages 39:1–39:6, 2015.

[27] P. Tamayo, C. Berger, M. Campos, J. Yarmus, B. Milenova, A. Mozes, M. Taft, M. Hornick, R. Krishnan, S. Thomas, M. Kelly, D. Mukhin, B. Haberstroh, S. Stephens, and J. Myczkowski. Oracle Data Mining. In O. Maimon and L. Rokach, editors, *Data Mining and Knowledge Discovery Handbook*, pages 1315–1329. Springer, 2005.

[28] M. Then, M. Kaufmann, A. Kemper, and T. Neumann. Evaluation of Parallel Graph Loading Techniques. In *GRADES 2016*, pages 4:1–4:6, 2016.

[29] M. Then, L. Passing, N. Hubig, S. Günnemann, A. Kemper, and T. Neumann. Effiziente Integration von Data-und Graph-Mining-Algorithmen in relationale Datenbanksysteme. In *Proc. LWA 2015 Workshops*, pages 45–49, 2015.

[30] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg. Top 10 Algorithms in Data Mining. *Knowledge and Information Systems*, 14(1):1–37, 2008.

[31] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proc. HotCloud 2010*, 2010.

Data Exploration with SQL using Machine Learning Techniques

Julien Cumin
Orange Labs & Inria
Meylan, France
julien1.cumin@orange.com

Jean-Marc Petit
Univ Lyon, INSA Lyon, CNRS
LIRIS, France
jean-marc.petit@liris.cnrs.fr

Vasile-Marian Scuturici
Univ Lyon, INSA Lyon, CNRS
LIRIS, France
marian.scuturici@liris.cnrs.fr

Sabina Surdu
Babes-Bolyai Univ
Cluj-Napoca, Romania
sabina@cs.ubbcluj.ro

ABSTRACT

Nowadays data scientists have access to gigantic data, many of them being accessible through SQL. Despite the inherent simplicity of SQL, writing *relevant and efficient SQL queries* is known to be difficult, especially for databases having a large number of attributes or meaningless attribute names. In this paper, we propose a "rewriting" technique to help data scientists formulate SQL queries, to rapidly and intuitively explore their big data, while keeping user input at a minimum, with no manual tuple specification or labeling. For a user specified query, we define a *negation query*, which produces tuples that are not wanted in the initial query's answer. Since there is an exponential number of such negation queries, we describe a pseudo-polynomial heuristic to pick the negation closest in size to the initial query, and construct a balanced learning set whose positive examples correspond to the results desired by analysts, and negative examples to those they do not want. The initial query is reformulated using machine learning techniques and a new query, more efficient and diverse, is obtained. We have implemented a prototype and conducted experiments on real-life datasets and synthetic query workloads to assess the scalability and precision of our proposition. A preliminary qualitative experiment conducted with astrophysicists is also described.

CCS Concepts

•Information systems → Data mining; Query reformulation; *Users and interactive retrieval;* **•Human-centered computing → Interactive systems and tools; •Computing methodologies →** *Machine learning;*

Keywords

Big Data; query rewriting; pattern mining; query recommendation; knowledge discovery in databases

1. INTRODUCTION

There's a pressing need on the discovery rate of interesting knowledge to keep pace with the booming data stores, as is the case in scientific fields like astronomy or earth observation, with their continuous advent of data-producing instruments and techniques. For instance, the Large Synoptic Survey Telescope [2] (LSST) currently under construction in Chile is projected to produce about 15 TB of imaging data each night, and over 10 petabytes of relational data in its 10 years lifetime. Clearly, our ability to stockpile data has long left behind our data analysis and exploitation capacities. Trying to make sense of data of the mentioned sizes with today's systems and techniques is an extremely difficult, if not an impossible, task, as even simple queries can return results so large they are practically incomprehensible [3]. Data storage and throughput related issues may be quite challenging, but problems linked with data exploration and system usage are even more so.

Scientific data commonly appear as SQL queryable relations, with hundreds of attributes holding numerical values from physical measurements or observations[1]. The attribute-based selection criteria are not always easily expressible. In order to formulate the right SQL query, analysts need to correctly define the appropriate attribute thresholds and make sure no attribute condition is missing from the query, which can be a real hurdle. Moreover, it is often the case that scientists do not know exactly what they are searching for, until after they find it, e.g., astrophysicists searching for interesting patterns in the data. Consider an astrophysicist exploring a huge database holding data about stars. She wants to select stars likely to harbor planets. But the data's attributes are mostly related to light magnitude and amplitude, and the presence of planets is only confirmed for a small number of stars. The astrophysicist can easily pose an initial query retrieving those stars but has no idea about another SQL query whose results would be similar, i.e., pro-

[1] See for example the LSST website: http://lsst-web.ncsa.illinois.edu/schema/index.php

viding a fair number of true positives (stars with planets) and a limited number of false positives (stars which surely lack planets), but also introducing new stars, which are *likely* to have planets.

Given the range of fields which produce or benefit from increasingly sized data, more and more difficult to query by a large spectrum of users, we believe it is crucial to enable data exploration, while keeping the expertise requirements as low as possible.

Example 1 Consider the `CA (CompromisedAccounts)` relation given in figure 1, inspired from the Ashley Madison data breach[2] [32]. Ashley Madison is a website targeting people that seek out an extramarital affair. `AccId` is a user identifier and `BossAccId` is the boss of the user, only if she is also registered online. `JobRating` is a job performance rating for some of the accounts and normalized to a scale of 1 to 5. `MoneySpent` and `DailyOnlineTime` refer to the money spent on the site, and the average daily time spent online by the owner's account, respectively. `Status` tells whether the user is a governmental employee or not. The meaning of the other attributes is clear from context.

Assume a zealous reporter is searching for *governmental users that spend more time online than their bosses.* The reporter has built up a somewhat good background in SQL, and poses the following query (initial query):

```
SELECT AccId, OwnerName, Sex
FROM CompromisedAccounts CA1
WHERE Status = 'gov' AND
 DailyOnlineTime > ANY
 (SELECT DailyOnlineTime
      FROM CompromisedAccounts CA2
      WHERE CA1.BossAccId = CA2.AccId)
```

This query produces the tuples corresponding to *Casanova* and *PrinceCharming.* But the reporter is interested in as many government officials as possible. Clearly, the problem of relaxing some conditions of the query to get more results has no solution for her. Hovewer, as we will see in the paper, the above query can be reformulated as follows:

```
Reformulated query:

   SELECT AccId, OwnerName, Sex
   FROM CompromisedAccounts
   WHERE (MoneySpent >= 90000 AND JobRating >= 4.5) OR
     (MoneySpent < 90000 AND DailyOnlineTime >= 9)
```

We notice that the reformulated query:

- is very much different from the initial one: it includes the new attributes *MoneySpent* and *JobRating*;
- it does not contain imbrications, therefore being more efficient, only going once through the relation; when the data volumes are very large, this is obviously of major importance;
- it maintains *Casanova* and *PrinceCharming* in the result, but
- is not equivalent to the first query, introducing some diversity in the results: tuples *RhetButtler*, *MrDarcy* and *BigBadWolf* are only found in the new query's result.

However, the reporter does not know in advance the conditions in the reformulation's *WHERE* clause, which in fact describe a *pattern* hidden in the data. According to this pattern, accounts spending more than 90k and whose owners do very well at work are likely to belong to cheating governmental employees who spend more time on the website than their bosses do. Same can be assumed about accounts spending less than 90k, but more than one third of a day online. A set of attributes is described based on another set of attributes. We could uncover this pattern by feeding a data mining tool with all these data, then go back to the database and pose the found query. But this poor reporter already deserves all the credit and our sympathy for digging into databases, a field unfamiliar to her. Do we really want to put her through the wringer of delving into the fascinating, yet complex world of data mining and machine learning? Moreover, if on a 10 rows example one could come up with a pattern close to the real one, this is impossible to achieve in a realistic setting.

Problem statement.

The problem we are interested in can be formulated as follows:

> Given a database and a user-specified SQL query that selects data based on some initial condition, we are interested in a reformulation of the initial query that captures the *pattern* revealed from the initial query's result data.

We do not search for an equivalent query, but for one whose result overlaps the initial query's result to a certain extent, and that also introduces new tuples in its result. The latter represents the exploratory potential of the new query. We set out to build an interaction that allows analysts to explore their data in this fashion, solely through SQL queries.

For a user-specified SQL query Q on a database d, we can easily define *positive examples*, or simply *examples*, based on Q's answer on d. They correspond to tuples that are wanted by the analyst. One of the challenges is to define the *negative examples* or *counter-examples*, i.e., tuples the analyst does not want to see in the result. This, in turn, raises the question of defining a query $\overline{Q}$ that could be considered as a *negation* of Q. If the generation of this query $\overline{Q}$ is possible, we have a set of *example* tuples, the results of Q's evaluation, and a set of *counter-example* tuples, obtained from $\overline{Q}$'s evaluation. We can then use a data mining technique in a supervised pattern learning approach on these two tuple sets.

For a user-specified query Q on a database d, there is an exponential number of possible negation queries $\overline{Q}$ that can be defined by negating different parts of Q's selection condition. We aim to find the one whose answer size is closest to Q's answer size, i.e., the set of positive examples and the set of negative examples are as close as possible in size. We call it the *balanced negation query.* The "more" balanced the learning set is, the higher its entropy, the better

[2] This is an example we came up with, in no way related to real data from the breach.

AccId	Owner Name	Age	Sex	Money Spent	Daily Online Time	JobRating	Status	BossAccId
100	Casanova	50	M	100k	5h	4.5	gov	350
200	DonJuanDeMarco	20	M	20k	1h	2.1	null	null
350	PrinceCharming	28	M	90k	4h	4.8	gov	230
40	Playboy	40	M	10k	35min	2	nongov	700
700	Romeo	50	M	30k	30min	3	nongov	null
90	RhetButtler	40	M	95k	4h	4.9	null	null
80	Shrek	40	M	25k	1h	1	nongov	700
70	MrDarcy	35	M	97k	3h	4.6	null	null
230	JackSparrow	61	M	30k	2h	3	gov	null
59	BigBadWolf	31	M	70k	9h	3	null	200

Figure 1: The `CompromisedAccounts (CA)` relation

for the decision tree algorithm working on it. Pruning the exponential space of negation queries boils down to solving an heuristic based on the Knapsack problem, known to be pseudo-polynomial.

The decision-tree algorithm automatically proposes a new query to the data analyst. This system-generated query can be interesting in the exploratory quest, since its results are close to, yet different from the initial query's results, including new tuples that were not directly accessible before. We propose some quality measures to evaluate the reformulated query quality, especially with respect to its diversity in terms of returned tuples.

We emphasize that:

- the user's input is kept to a minimum, i.e., only a SQL query is expected from her; she does not need to manually label any tuples, or input actual tuple values, an approach frequently used by alternative systems;
- even for users with a data mining background, the job is now much easier, since they only use one system, without any disrupting switches between various tools.

Paper contribution.

To sum up, we propose a simple but powerful approach to deal with the above problem statement. Our main contributions can be summarized as follows:

- For a class of SQL queries, we introduce the notions of *positive examples* - from the answer set of a query - and *negative examples* - from the answer set of a *negation query*.
- Since the set of negation queries is exponential, we propose a pseudo-polynomial Knapsack-based heuristic to identify a negation query whose size is close to the size of the initial query answer.
- The initial query and the selected negation query allow building a learning set on which machine learning techniques, derived from decision trees, are applied. A new SQL query, called a transmuted query, is proposed from the decision tree. Some criteria are proposed to evaluate the quality of the transmuted query.
- We have implemented a prototype and conducted experiments on real-life datasets and synthetic query workloads to assess the scalability and precision of our proposition. A preliminary qualitative experiment conducted with astrophysicists is also described.

Paper Organization.

The rest of this paper is organized as follows. Section 2 discusses queries and their *negation queries*. A heuristic that prunes the exponential space of negation queries is fully described. In section 3 we present our approach to data exploration, based on automatic query rewriting. We discuss *transmuted queries* and metrics that assess the quality of the rewriting. Section 4 describes preliminary results on an astrophysics database and the experimental setting for a newly implemented prototype. Section 5 positions our approach with respect to related work. Finally, section 6 concludes our paper.

2. SQL QUERIES AND THEIR NEGATION

The challenge we take on is to define the set of tuples that *do not verify* a query Q, which reduces to defining a *negation query* $\overline{Q}$ of Q. We then explore the exponential space of possible negation queries, and finally present a pseudo-polynomial Knapsack-based heuristic to find the negation query whose answer size is closest to the answer size of Q.

2.1 Preliminaries

We briefly introduce the notations used throughout this paper (see for example [4]). Let d be a database defined on a schema $\mathbf{R}$ and Q be a query on $\mathbf{R}$. We denote by $ans(Q, d)$ the result of the evaluation of Q on d. The domains of attributes are assumed to give either categorical values or numerical values. We assume the database allows null values. As relational languages, we consider both relational algebra for formal notations and SQL for the running example.

2.2 Considered relational queries

For the sake of clarity, we consider a simple class of relational queries, basically conjunctive queries extended with some binary operators and a construct to check for NULL values. Identifying a larger class of relational queries is left for future work.

Queries in the considered class have the following form:

$$Q = \pi_{A_1,\ldots,A_n}(\sigma_F(R_1 \bowtie \ldots \bowtie R_p))$$

where

- π is the projection, σ the selection and $\bowtie$ the natural join as usual
- F is a conjunction of m atomic formulas of the form $\gamma_1 \wedge \ldots \wedge \gamma_m$ with $m \geq 1$

- Each atomic formula (or predicate) γ_i in F has the form $A\ bop\ B$, $A\ bop\ a$, $A\ IS\ NULL$, where A, B are from R, a is a real value or a categorical one and $bop \in \{=, <, >, \leq, \geq\}$
- An atomic formula γ can be negated; its negation is denoted by $\neg(\gamma)$.

We denote by $attr(F)$ the set of attributes that appear in a selection formula F. The cardinal of a set E is denoted by $|E|$.

Example 2 To comply with this class of queries, the initial query from example 1 is rewritten as:

```
SELECT CA1.AccId, CA1.OwnerName, CA1.Sex
FROM CompromisedAccounts CA1, CompromisedAccounts CA2
WHERE CA1.Status = 'gov' AND
  CA1.DailyOnlineTime > CA2.DailyOnlineTime AND
  CA1.BossAccId = CA2.AccId
```

For a query Q of the above form, we consider the "reservoir of diversity" (diversity tank) to consist of those tuples for which:

- there exists at least one predicate γ_i whose evaluation is NULL; (1)
- all the predicates in Q that do not evaluate to NULL, evaluate to TRUE (2).

Some tuples in this diversity tank might turn out to be interesting for the user, even if they do not appear in the initial query's result. Condition (1) states that, for a tuple to have an exploratory potential, some data the user is interested in about the tuple needs to be unknown. Condition (2) asks that none of Q's predicates evaluate to false. Otherwise, the tuple does not satisfy Q and shouldn't be explored at all, since the user is interested in tuples meeting Q's condition.

Example 3 For our running example, tuples corresponding to employees (CA1.OwnerName) *DonJuanDeMarco*, *RhetButtler*, *MrDarcy*, *JackSparrow* and *BigBadWolf* form the diversity tank (and indeed, the reformulated query's new tuples, *RhetButtler*, *MrDarcy* and *BigBadWolf*, came from this set).

Positive examples are denoted by $E_+(Q)$ and come from the query's evaluation result. When the result's size is reasonable, we can consider all its tuples as examples. Otherwise, we can use stratified random sampling for instance to extract a subset of tuples as positive examples. We keep all the possible attributes, so later on in the learning phase we have as many as possible options to learn on. Therefore, we eliminate the projection on $A_1, ..., A_n$ and obtain: $E_+(Q) \subseteq \sigma_F(R_1 \bowtie \ldots \bowtie R_p)$.

Example 4 The positive tuples of query Q are those corresponding to employees *Casanova* and *PrinceCharming*.

2.3 The negation of queries

Let Q be a query. Its answer set is obtained by simply evaluating Q on the database. We pose the problem of defining $\overline{Q}$, a *negation query* of Q, i.e., a query whose evaluation produces tuples we do not want to see when evaluating Q. Intuitively, $\overline{Q}$ should not touch the same tuples of Q, i.e., the intersection of their respective answer sets should be empty.

Since we aim to uncover dependencies between attributes in the query rewriting process, we keep all the possible attributes that might allow us to distinguish between tuples in the learning step. As for positive examples, we eliminate the projection from the negation queries' definitions hereafter.

One way of computing a negation query for Q is to consider its entire tuple space $R_1 \bowtie \ldots \bowtie R_p$ and eliminate those tuples that belong to Q's answer. This is in fact the complement of Q. We consider it to be Q's *complete negation* and denote it by $\overline{Q_c}$:

$$\overline{Q_c} = (R_1 \bowtie \ldots \bowtie R_p) \setminus (\sigma_{\gamma_1 \wedge \ldots \wedge \gamma_m}(R_1 \bowtie \ldots \bowtie R_p)) \quad (1)$$

There is no guarantee on the answer size of $\overline{Q_c}$; it may be quite different than Q's size. We thus set out to explore the space of alternative negation queries for Q. We consider negation queries whose selection conditions are generated from Q's selection formula. In this case, we consider that a negation query $\overline{Q}$ needs to negate at least one of Q's predicates.

We exclude foreign key join predicates from the set of predicates that can be negated. They help narrow down the set of tuples that can be used as examples and counterexamples.

To simplify notations, we describe Q's selection formula F as $F_{\overline{k}} \wedge F_k$, where F_k is the conjunction of all the foreign key predicates in F, if any, and $F_{\overline{k}}$ the conjunction of all the other predicates, i.e., the *negatable* predicates.

Property 1 Let Q be a query and $n = |F_{\overline{k}}|$, the number of negatable predicates. The number of negation queries $\overline{Q}$ with respect to Q is exponential in n.

Proof. For each negatable predicate γ in Q, there are three possibilities: (1) keep it in $\overline{Q}$ as it is, (2) take its negation $\neg(\gamma)$ or (3) do not consider it at all. Then there are 3^n possible negation queries, but 2^n out of them are invalid, since they do not contain any negated predicate from $F_{\overline{k}}$. We get $3^n - 2^n$, which is in $O(2^n)$. □

We denote the set of the valid negation queries by $\{\overline{Q}\}$. All the negated γ_i come from $F_{\overline{k}}$. We denote by $attr(\overline{F_{\overline{k}}})$ all the attributes from $F_{\overline{k}}$ that appear in predicates that are negated in $\overline{Q}$.

Similarly to positive examples, the negative examples are denoted by $E_-(Q)$ and verify the following:
$E_-(Q) \subseteq ans(\overline{Q}, d)$.

Example 5 Let us denote the three predicates in Q CA1.Status = 'gov' by γ_1, CA1.DailyOnlineTime > CA2.DailyOnlineTime by γ_2 and CA1.BossAccId = CA2.AccId by γ_3. In our approach there are five possible negations of Q, with selection formulas: $\neg(\gamma_1) \wedge \gamma_3$, $\neg(\gamma_2) \wedge \gamma_3$, $\neg(\gamma_1) \wedge \gamma_2 \wedge \gamma_3$, $\gamma_1 \wedge \neg(\gamma_2) \wedge \gamma_3$ and $\neg(\gamma_1) \wedge \neg(\gamma_2) \wedge \gamma_3$. If we choose the third one, the negation query $\overline{Q}$ is:

```
SELECT *
FROM CompromisedAccounts CA1, CompromisedAccounts CA2
WHERE NOT (CA1.Status = 'gov') AND
  CA1.DailyOnlineTime > CA2.DailyOnlineTime AND
  CA1.BossAccId = CA2.AccId
```

Accounts *Playboy* and *Shrek* are thus considered to be negative examples. Both Q and $\overline{Q}$ answer sizes are equal to two.

2.4 Pruning the space of negation queries

In this subsection we solve the problem of finding the closest negation query, in terms of its answer size, for a given query, by proposing an heuristic based on the subset-sum problem (related to the knapsack problem). The subset-sum problem is known to be pseudo-polynomial (weakly NP-complete) [7]. The additional constraints we add do not affect the complexity of the algorithm.

Notation.

To further simplify notations, when there's no confusion between a query Q and its answer, we denote the latter by Q instead of $ans(Q, d)$. Z denotes the entire tuple space $R_1 \bowtie \ldots \bowtie R_p$. Similarly, for a predicate $\gamma_i \in Q$, we refer to the query $\sigma_{\gamma_i}(Z)$ simply by γ_i. Since we only touch the selection formula, we ignore Q's projection attributes. We denote the negation of a predicate γ by $\overline{\gamma}$.

Assumption.

DataBase Management Systems (DBMS) maintain many statistics for cost-based optimization of query processing. Moreover, to estimate the size of query results and make a decision for choosing a physical plan, data are often assumed to be uniformly distributed. In the sequel, we borrow the same assumptions, i.e., data is uniformly distributed in Z and for a given query Q, an estimate of the size of its answer $|Q|$ is supposed to be known. We denote the probability that a tuple in Z verifies γ_i by $P(\gamma_i)$. The cardinality of γ_i is $|\gamma_i| \simeq P(\gamma_i) * |Z|$. The probability that a tuple satisfies both γ_i and γ_j is $P(\gamma_i \wedge \gamma_j) = P(\gamma_i) * P(\gamma_j)$. The cardinality of the set of rows satisfying both predicates is estimated by $|\gamma_i \wedge \gamma_j| \simeq P(\gamma_i) * P(\gamma_j) * |Z|$. The probability of a negated predicate $\overline{\gamma_i}$ is $P(\overline{\gamma_i}) = 1 - P(\gamma_i)$, and $P(Q \cup \overline{Q_c}) = 1$.

We can now give a more formal description of our problem of finding the most balanced learning set corresponding to a query Q.

Balanced negation query.

Let us consider a query Q with n negatable predicates in $F_{\overline{k}}$ and l foreign key join predicates in F_k. The problem statement is the following:

> Given such a query Q, find a negation query $\overline{Q}$ of Q such that:
>
> (1) its answer size $|\overline{Q}|$ is closest to $|Q|$, i.e., $abs(|Q| - |\overline{Q}|)$ is minimized,
>
> (2) $\overline{Q}$ negates at least one predicate from $F_{\overline{k}}$,
>
> (3) $\overline{Q}$ can contain any number of the rest of the predicates from $F_{\overline{k}}$, negated or not, and
>
> (4) $\overline{Q}$ contains all the predicates from F_k.

The possible solutions to this problem have the form $\overline{Q} = \bigwedge_{i=1}^{n+l} a_i$, where a_i is a predicate. Let us consider the predicates for $\overline{Q}$ as a (n+l)-tuple denoted by s. The components of s are:

(1) the predicates from F_k,

(2) any predicate from $F_{\overline{k}}$, or its negation, or the "identity" element $Q \cup \overline{Q_c}$, i.e., the predicate is not considered at all in $\overline{Q}$.

s can be represented as: $s = (a_1, \ldots, a_{n+l}) = (\gamma_{k_1}, \ldots, \gamma_{k_l}, e_1, \ldots, e_n)$ with $\gamma_{k_i} \in F_k$ and $e_j \in \left\{\gamma_{\overline{k}_j}, \overline{\gamma_{\overline{k}_j}}, Q \cup \overline{Q_c}\right\}$, for all $\gamma_{\overline{k}_j} \in F_{\overline{k}}$.

Therefore every negation query $\overline{Q}$ can be represented as such a tuple s and its cardinality is estimated by $|\overline{Q}| = |s| = \left(\prod_{i=1}^{n+l} P(a_i)\right) * |Z| = \left(\prod_{i=1}^{n+l} \frac{|a_i|}{|Z|}\right) * |Z|$.

Our problem now is in fact a particular case of the *subset product problem* [19, 26], known to be NP-hard [24]:

> Given a set $E = \{e_1, \ldots, e_n\}$ of integer values and a number m, does there exist a subset F of E, such that $\prod_{e_i \in F} e_i = m$?

For a given query Q, we need to choose all the predicates in F_k and one of three possible versions for the predicates in $F_{\overline{k}}$, such that the product of their probabilities multiplied by the cardinal of the tuple space Z is as close as possible to the size of Q, the target number.

Applying logarithms on this product, we pass from the subset-product problem to a kind of subset-sum problem on real numbers (the pseudo-polynomial algorithm is working on integers).

$$\sum_{e_i \in F} log(e_i) = log(m)$$

We propose a heuristic by introducing a tolerable approximation in the precedent relation:

$$\sum_{e_i \in F} \lfloor log(e_i) * sf \rfloor = \lfloor log(m) * sf \rfloor$$

where the *scale factor* $sf \geq 1$ is used to reduce the approximation due to rounding logarithms.

Our problem can now be expressed as a particular case of the subset-sum problem, also known to be NP-complete:

> Given a total weight $|Q|$ and a set of $n + l$ a_i objects, l of them with fixed weights $|\gamma_{k_i}|$, $i = 1, \ldots, l$, and n of them with three possible weights $|e_j| \in \left\{|\gamma_{\overline{k}_j}|, |\overline{\gamma_{\overline{k}_j}}|, 1\right\}$, $j = 1, \ldots, n$, choose all the γ_{k_i} objects and a version for **each** of the e_j objects, such that: (1) the sum of the combined weights of the chosen objects is as close as possible to $|Q|$ and (2) at least one of the e_j objects is negated.

If s is the solution tuple, its combined weight is its cardinality as defined above, and condition (1) translates to minimizing $abs(|Q| - |s|)$.

The heuristic described in algorithm 1 solves this problem. All the objects in the fixed-weights vector f_k are in the output; the target weight w is updated accordingly (lines 2-3). Function *weight* calculates the weight of s, i.e., the number of rows estimated to satisfy all the predicates currently in s. From here on out, we only work with the $f_{\overline{k}}$ vector of negatable objects / predicates. For each such predicate (weight: $lWs[i].pW$), BALANCEDNEGATION assumes its negation (weight: $lWs[i].nW$) is part of the solution. It finds an optimal solution for the rest of the predicates and a correspondingly updated weight (lines 5 - 15). This solution could contain only positive objects, but the current predicate is always added with its negated form, so restriction (2) in the problem statement is met. Out of the n possible solutions, the one that gets closer to the target weight is chosen. We set the scale factor sf parameter value to 1000 (see experiment 2 in section 4.1).

Algorithm 1: Knapsack-based heuristic to find the balanced negation of a query

```
1  procedure BalancedNegation ($|Z|, |Q|, f_k, f_{\overline{k}}, sf$) ;
     Input  : the size of the tuple space $|Z|$; the target
              weight $|Q|$; $l$-vector of fixed weights $f_k$;
              $n$-vector of negatable weights $f_{\overline{k}}$; scale factor
              $sf$
     Output: $l + n$-vector of chosen objects $s$; $s$'s weight $s_w$
2  $s := f_k; s_w := weight(s)$ ;
3  $w := \frac{|Q|}{s_w}$ ;
4  $mW := 0$ ;
5  for $i \leftarrow 1$ to $n$ do
6      $lWs := f_{\overline{k}}$ ;
7      $rW := lWs[i].pW$ ;
8      $lWs.Remove(i)$ ;
9      $tW := \frac{w*|Z|}{|Z|-rW}$ ;
10     $tW := -\left\lfloor \ln(\frac{tW}{|Z|}) * sf \right\rfloor$ ;
11     for $j \leftarrow 1$ to $n-1$ do
12         $lWs[j].pW := -\left\lfloor \ln(\frac{lWs[j].pW}{|Z|}) * sf \right\rfloor$ ;
13         $lWs[j].nW := -\left\lfloor \ln(\frac{|Z|-lWs[j].pW}{|Z|}) * sf \right\rfloor$ ;
14     end
15     $SubsetSum(lWs, tW, out\ oObj, out\ oW)$ ;
16     $oW := \left\lfloor e^{\frac{(-oW)}{sf}} * |Z| \right\rfloor$ ;
17     $mWL := \left\lfloor \frac{|Z|-rW}{|Z|} * oW \right\rfloor$ ;
18     if $mWL > mW$ then
19         $mW := mWL$ ;
20         $CompleteSol(i, s, s_w, oObj, mWL)$ ;
21     end
22 end
```

In lines 10-14 **BalancedNegation** transforms the input in order to apply a classic Knapsack algorithm on it. For each predicate, the algorithm computes its probability and then logarithms it. Since probabilities are subunitary, all the logarithms are negative and quite small, so they are multiplied by the scale factor sf, and the opposite of their integer part is retained. The same treatment is applied to the target weight. SubsetSum in line 15 computes an optimal solution in vector $oObj$ with the corresponding sum of the solution subset in $oW \leq tW$. The only difference from the classic algorithm is that, if object $lWs[i]$'s positive version $lWs[i].pW$ is chosen, then its negation can not be part of the solution and vice-versa.

The output weight oW is transformed back in line 16. Lines 17-21 choose the best out of the n possible solutions adding the temporarily removed object at position i with its negated version ($CompleteSol$).

3. AUTOMATIC QUERY REWRITING

We now describe the machine learning stage that uncovers relevant patterns in the data. Starting from a user's initial query Q, the system automatically generates its negation $\overline{Q}$. It then assembles a learning set from $E_+(Q)$ and $E_-(Q)$, which is fed into an implementation of the C4.5 decision tree learning algorithm. The output decision tree's patterns are forthrightly translated into a relational selection condition, yielding a new SQL query, the transmuted query of Q. Different criteria evaluating the quality of our machine learning-based rewriting approach are defined.

3.1 Learning set construction

Once we have the positive and the negative example sets, we can build a learning set. As stated before, we do not aim at producing a reformulated query that provides the exact, precise answer of the initial query. Instead, we want to help the analyst formulate a query that better answers his expectations. These guesswork stages do not require the consideration of all the data if its size is very large. A detailed study of the guarantees we can provide for learning is outside the scope of this paper.

Definition 1 Given a query Q and its negation $\overline{Q}$, a *learning set* is defined on the schema of $(R_1 \bowtie \ldots \bowtie R_p) \setminus attr(\overline{F_{\overline{k}}}) \cup Class$. Its tuples come from $E_+(Q)$ and $E_-(Q)$, with the addition of the $+$, and the $-$ value, respectively, for the new `Class` attribute.

We exclude attributes in $attr(\overline{F_{\overline{k}}})$ from the learning set's schema to avoid learning (part of) the selection condition expressed in the initial query.

Example 6 Going further with the running example, we obtain the learning set described in Figure 2. The `Status` attribute, i.e., the only attribute in $attr(\overline{F_{\overline{k}}})$, has been suppressed and the `Class` attribute has been added with the corresponding $+$ and $-$ values.

Decision tree-based machine learning methods construct a tree that determines a class variable as a function of input variable values. A path from its root to a leaf forms a conjunction of conditions on the input variables. The class of the data that fall through this path is given by the label of its leaf. The supervised model construction, i.e., tree structure and conditions placed on internal nodes, based on a learning set, depends on the used algorithms. We chose the C4.5 algorithm [29]. It allows us to predict the values of the *Class* attribute depending on a set of attributes from the learning set.

3.2 Building the selection condition from a decision tree

Starting from a decision tree, it's relatively straightforward to build a relational selection condition by traversing the tree in depth. A branch in a tree is a direct path from its root to a leaf. A branch in the decision tree is a conjunction of boolean conditions on a tuple's attributes values. The class of the tuple is the label on the leaf of the branch. The set of branches leading to the class of positive tuples "+" can thus be seen as a disjunction of conjunctive clauses obtained from the branches. This disjunction can hence be used like a new relational selection condition.

CA	AccId	Owner Name	Age	Sex	Money Spent	Daily Online Time	JobRating	BossAccId	Class
	100	Casanova	50	M	100k	5h	4.5	350	+
	350	PrinceCharming	28	M	90k	4h	4.8	230	+
	40	Playboy	40	M	10k	35min	2	700	-
	80	Shrek	40	M	25k	1h	1	700	-

Figure 2: Learning set built from $E_+(Q)$ and $E_-(Q)$

Definition 2 Let *lSet* be a learning set obtained from a query Q. Let *decisionTree* be the decision tree learned from *lSet* to predict the values of the *Class* attribute. Let b be a positive branch of *decisionTree*, i.e., from the root to a leaf labeled +. We use the following notation for the disjunction of conjunctions leading to positively labeled leaves:

$$F_{new} = \bigvee_{b \in decisionTree} \bigwedge_{e \in b} e$$

where e has the form $A_i \, bop \, v$, A_i is an attribute in *lSet*, *bop* is a usual binary operator, and v is a numerical or categorical value.

Definition 3 Let Q be a query. The rewritten query obtained from Q, denoted by ${}^tQ$, is defined as follows:

$${}^tQ = \pi_{A_1,\ldots,A_n}(\sigma_{F_{new}}(R_1 \bowtie \ldots \bowtie R_p)).$$

We call it *the transmuted query* thereafter.

The transmuted query has a completely new selection condition that can include attributes not identified as useful in the initial query. Moreover, ${}^tQ$ is obviously simpler and quicker if the initial query is nested with, for example, the `EXISTS` or *bop* `ANY` operators. The rewriting is mechanically simplified by a single selection (a single data scan only).

Example 7 In the running example, a decision tree algorithm finds the condition `(MoneySpent >= 90000 AND JobRating >= 4.5) OR (MoneySpent < 90000 AND DailyOnlineTime >= 9)` . The corresponding rewritten query ${}^tQ$ is:

```
SELECT AccId, OwnerName, Sex
FROM CompromisedAccounts
WHERE (MoneySpent >= 90000 AND JobRating >= 4.5) OR
  (MoneySpent < 90000 AND DailyOnlineTime >= 9)
```

3.3 Quality criteria

It is difficult to make guarantees about the precise relationships between the initial query Q and its rewriting ${}^tQ$, since the latter depends on the patterns discovered in the learning phase. We can however describe the sets of tuples involved in the rewriting process and give their optimal properties:

- $\boldsymbol{Z}$: the entire tuple space $R_1 \bowtie \ldots \bowtie R_p$
- tuples fulfilling F_k and tuples fulfilling $F_{\overline{k}}$
- $\boldsymbol{ans(Q, d)}$: the set of tuples from the initial query Q on d, i.e., tuples that meet both F_k and $F_{\overline{k}}$
- $\boldsymbol{ans(\overline{Q}, d)}$: the set of tuples from the negation $\overline{Q}$ of Q on d, i.e., tuples that meet F_k, but do not meet $F_{\overline{k}}$
- $\boldsymbol{E_+(Q)}$: the set of positive tuples
- $\boldsymbol{E_-(Q)}$: the set of negative tuples
- $\boldsymbol{ans({}^tQ, d)}$: the set of tuples from the new query ${}^tQ$ on d

We now explicitly define a number of criteria and metrics that assess the quality of ${}^tQ$, the query obtained from Q.

3.3.1 Representativeness of the initial data

Our objective is to obtain a query ${}^tQ$ whose evaluation is representative of Q's results. Since the supervised learning process uses examples and counter-examples, we can expect to obtain patterns that help meet this criterion. We concretely measure it with the following formulas:

$$\frac{|{}^tQ \cap Q|}{|Q|} \underset{\text{optimal}}{=} 1 \tag{2}$$

$$\frac{|{}^tQ \cap \pi_{A_1,\ldots,A_n}(\overline{Q})|}{|\pi_{A_1,\ldots,A_n}(\overline{Q})|} \underset{\text{optimal}}{=} 0 \tag{3}$$

Equation 2 justifies the direct representativeness of the data obtained from ${}^tQ$ with respect to the data obtained from Q: optimally, we should find in ${}^tQ$ all the tuples of Q. In a similar manner, equation 3 evaluates the proportion of tuples from $\overline{Q}$ found in ${}^tQ$, which should be as small as possible.

Example 8 Criteria 2 and 3 are optimal for the transmuted query in example 7. Indeed, this query retrieves both positive tuples *Casanova* and *PrinceCharming*, and does not produce any of the negative tuples *Playboy* or *Shrek*.

3.3.2 Diversity with respect to the initial data

The objective of the rewriting process is to produce a query that is similar to an initial query specified by the user (measurable by the previous criterion), but that also answers the user's exploratory expectation, and thus, presents new tuples to the user. Therefore, it is important that this set of new tuples is not only not empty (equation 4), but also of a suitable size: not too small with respect to the data initially obtained by the user (equation 5), nor comparable to the size of the entire set of tuples (equation 6). If the last condition is not met, the user will most likely have difficulties interpreting the result.

$${}^tQ \cap (\pi_{A_1,\ldots,A_n}(Z) - (Q \cup \pi_{A_1,\ldots,A_n}(\overline{Q}))) \neq \emptyset \tag{4}$$

$$|{}^tQ \cap (\pi_{A_1,\ldots,A_n}(Z) - (Q \cup \pi_{A_1,\ldots,A_n}(\overline{Q})))| \not\ll |Q| \tag{5}$$

$$|{}^tQ \cap (\pi_{A_1,\ldots,A_n}(Z) - (Q \cup \pi_{A_1,\ldots,A_n}(\overline{Q})))| \ll |\pi_X(Z)| \tag{6}$$

Example 9 The rewritten query ${}^tQ$ from example 7 produces three new tuples *RhetButtler*, *MrDarcy* and *BigBadWolf*, so criterion 4 is fulfilled. These three tuples are numerically comparable with respect to the 2 initial tuples (5), and are less numerous than the ten possible tuples (6).

We give the compact representation of the query rewriting approach in algorithm 2.

Algorithm 2: query rewriting

```
procedure QueryRewriting $(Q, d)$ ;
Input  : a query $Q$, database d
Output: the transmuted query ${}^tQ$
$let\ Q = \pi_{A_1,\dots,A_n}(\sigma_{F_k \wedge F_{\overline{k}}}(R_1 \bowtie \dots \bowtie R_p))$ ;
$SplitInTrainingAndTestSets(d, out\ trSet, out\ teSet)$ ;
$E_+(Q) := EvaluateQuery(Q, trSet)$;
$\overline{Q} := BalancedNegation(|trSet|, |Q|, F_k, F_{\overline{k}}, 1000)$ ;
$E_-(Q) := EvaluateQuery(\overline{Q}, trSet)$ ;
$lSet := BuildLearningSet(E_+(Q), E_-(Q), attr(\overline{F_{\overline{k}}}))$ ;
$decisionTree := FindC45(lSet)$ ;
$F_{new} := \bigvee_{b \in decisionTree_+} \bigwedge_{e \in b} e$ ;
return ${}^tQ = \pi_{A_1,\dots,A_n}(\sigma_{F_{new}}(R_1 \bowtie \dots \bowtie R_p))$ ;
```

4. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We have implemented the proposition made in this paper in `C#` and `SQL Server`, with the C45Learning classifier in the `Accord.NET` library [1], which implements the C4.5 algorithm [29]. We have conducted experiments first to evaluate the accuracy of the generated negation query with respect to an optimal negation query and second, to study the efficiency of the generation of negation queries. Based on previous results [13], we also have briefly described a validation of SQL data exploration with Astrophysicists on a real-life example.

4.1 Scalability and precision

Our proposition makes use of several existing pieces of work (e.g. decision tree, query evaluation) which are not detailed here. We focus on the most difficult part of our proposition, i.e., the heuristic to identify a negation query whose result size is similar to the size of the answer set of the initial query. Quality criteria detailed in section 3.3 require a cohort of users to assess the results and will be addressed in future work.

For a given query, our Knapsack-based heuristic is evaluated with respect to its *accuracy*, how good is the result with respect to the best possible negation query and its *efficiency*.

Both accuracy and efficiency have been studied with respect to different query workloads, a varying number of predicates in the initial SQL query and the values of the `sf` parameter.

Experimental setup.

In our experiments we used two datasets:

- `Iris`: a small well-known dataset describing the properties of some species of iris flowers. The dataset has 150 tuples only, four numerical attributes and one categorical attribute. It was chosen to easily compute (and understand) all the possible negation queries for a given query.
- `Exodata`: a scientific dataset containing 97717 tuples and 62 attributes (see next section for more details).

We generated a query workload as follows: for a fixed number of predicates in a query (from 1 to 200), a predicate of the form *A bop value* is generated by randomly choosing an attribute A, the operator *bop* from a list of possibilities ($\{=\}$ for categorical attributes, $\{<, <=, >, >=\}$ for numerical attributes), and the corresponding value $value \in Dom(A)$ for attribute A.

We assume to have statistics available for each attribute in the database, and hence the size of the database does not interfere with the performance of the algorithm.

To assess the distance between the negation proposed by our heuristic and the "best negation" for a query, we proceeded as follows: for a given query Q, we computed all its negations $\overline{Q}$; it has been indeed possible since the number of predicates remained small on our workloads.

We denote by $\overline{Q}_T$ the negation query closest in size to Q and by $\overline{Q}_K$ our approximated negation of Q.

The **distance** between $\overline{Q}_K$ and $\overline{Q}_T$ is defined by

$$abs(|\overline{Q}_K| - |\overline{Q}_T|)/|Z|$$

where $|Z|$ is the size of all possible tuples. The closer the distance is to zero, the better the heuristic (and conversely, the closer to one, the worse the heuristic).

Experiment 1.

To evaluate the impact of the *number of predicates* on the **accuracy** and *computation time* of the approximated negation, we fixed the scale factor `sf`$= 1000$ and we generated a query workload of 10 random queries with 1 to 9 predicates.

A *query type* is defined by its number of predicates. For each query type, we processed 10 queries and we displayed several values: the minimum distance, the first quartile, the third quartile and the maximum distance via a box plot. The average values for the distance are also given.

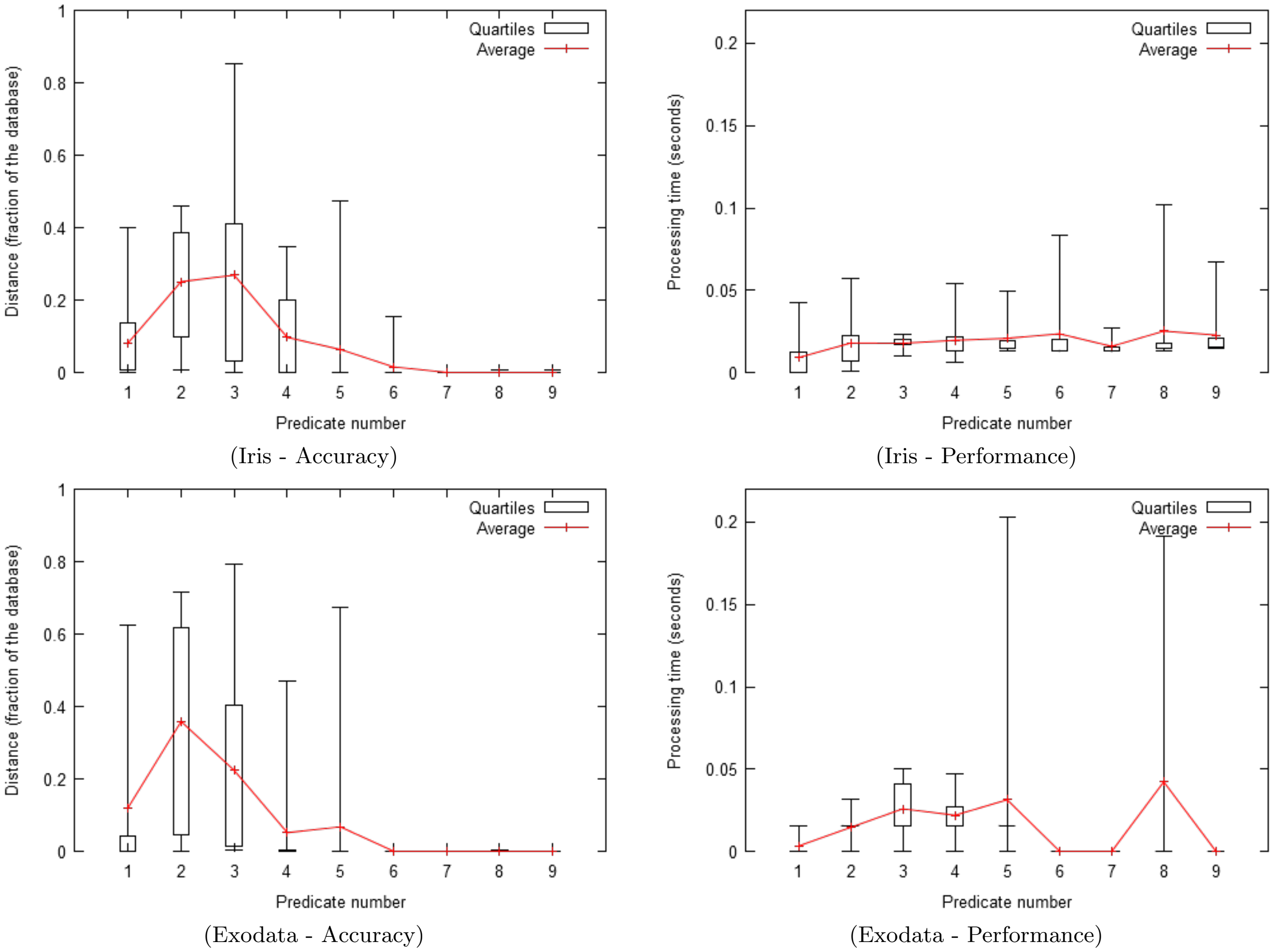
Figure 3 (top-left) shows the distance between the proposed heuristic and the closest negation (accuracy). For three predicates, we have a very bad result for one generated query (distance around 0.84) but, on average, the errors are around 0.2, which remains acceptable. The results show that the more predicates a query has, the better the heuristic is, very close to the best solution. With more than 6 predicates, the heuristic turns out to be very precise. Accuracy for both datasets is always excellent whenever the number of predicates exceeds six.

Figure 3 (top-right) shows good performances for the proposed heuristic on both datasets, always less than 0.2s.

Experiment 2.

The second test evaluates the impact of the **scale factor** sf on the **accuracy** of the approximated negation. The scale factor `sf` varies between 1 and 10000. As in the previous experiment, we used a query workload with 10 random queries for each type of test defined by the number of predicates (between 5 and 20) and different values of sf.

In our tests on *Exodata*, we observe that the accuracy is affected by different values of `sf`. For the same number of predicates, as the value of `sf` is increasing, the accuracy



(Iris - Accuracy)

(Iris - Performance)

(Exodata - Accuracy)

(Exodata - Performance)

Figure 3: Impact of the number of predicates on the accuracy and computation time of the approximated negation w.r.t. *Iris* dataset (top) and *Exodata* dataset (bottom).

is also improving. Whenever the value of `sf` exceeds 1000, the heuristic behaves very well (distance gets closer to 0, see figure 4-left).

Experiment 3.
As expected, the scale factor `sf` has an influence on the processing time. As seen in experiment 2, a greater value for `sf` allows a better approximation for the negation in our heuristic, but the search space increases. For a large number of predicates we execute the same tests only on the `Exodata` schema in order to estimate the overhead introduced by our heuristic in computing the negation for a given query. We observe that the processing time is increasing with the number of predicates in the original query and the value of `sf` (figure 4-right). However the processing time remains around 1 second for a query with 200 predicates and $sf = 10000$.

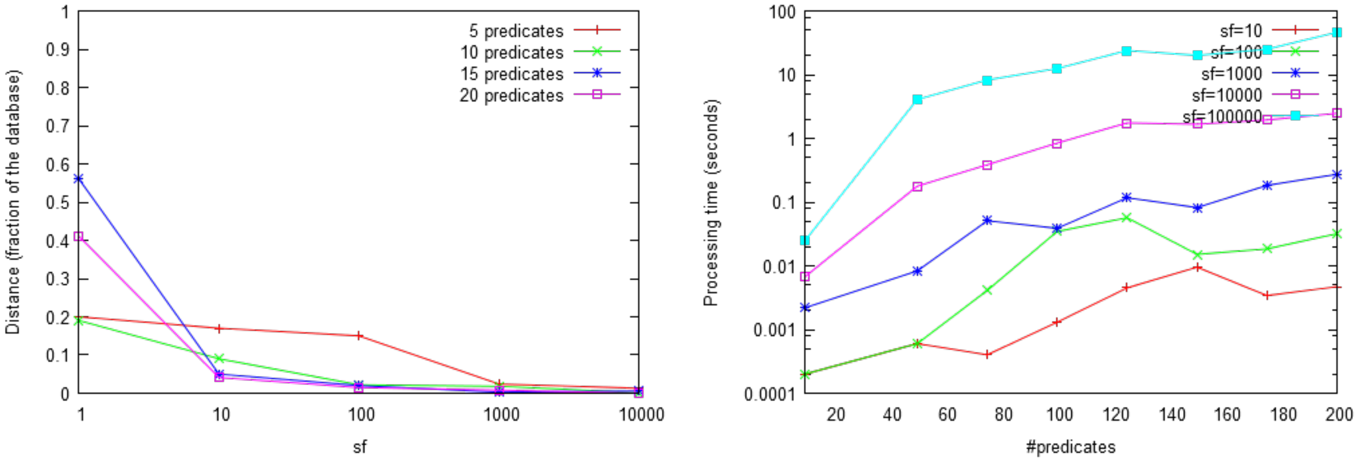
4.2 Validation with astrophysicists

We describe the validation conducted on an astrophysics database derived from the European project `CoRoT`[3] (COnvection, ROtation and planetary Transits), which studies star seismology and searches for extra-solar planets. `CoRoT` has observed for years the stars in our galaxy in order to study them and to discover planets beyond our solar system. A sample of the `EXODATA`[4] database was extracted into one table (`EXOPL`) with 97717 tuples and 62 attributes. A tuple represented a star and attributes included the position of the star, its magnitude at different wavelengths, the degree of its activity, etc. A special attribute `Object` described the presence of planets around the star - value `p`, the absence of planets - value `E`, or the lack of knowledge concerning possible revolving planets - `NULL`.

Most of the stars had not been classified, having the `NULL` value for the `Object` attribute. The objective was to obtain a set of stars that potentially harbor planets. We wanted to identify some conditions that allow to infer the presence of planets for stars that have not yet been studied, starting from the stars for which the presence or absence of planets has been confirmed. With astrophysicists, the initial query was thus very simple to identify:

[3] http://smsc.cnes.fr/COROT/

[4] http://cesam.lam.fr/exodat



(Queries on Exodata) (Queries on Exodata schema)

Figure 4: Impact of the scale factor sf on the accuracy of the approximated negation w.r.t. ***Exodata*** **dataset (left) and the computation time overhead needed to find this negation on** ***Exodata*** **schema (right).**

```
SELECT DEC, FLAG, MAG_V, MAG_B, MAG_U
FROM EXOPL
WHERE OBJECT = 'p'
```

The negation query was straightforward to obtain (without the machinery introduced in this paper), simply by changing the condition to `OBJECT = 'E'` (see [13] for a detailed discussion). There were 50 positive examples (`OBJECT = 'p'`) and 175 counter-examples (`OBJECT = 'E'`) among the 97717 tuples in the database. Discussions with astrophysicists emphasized different magnitude and amplitude attributes as pertinent attributes to learn on. They hold data concerning observed light under a variety of wavelength filters. Starting from this expert but easily exploitable information, we tried out, in a couple of minutes, several sets of attributes on which to learn. The expert selected attributes `MAG_B, AMP11, AMP12, AMP12, AMP13` and `AMP14`. The learning phase was then launched and generated a decision tree from which the following new condition was extracted: `MAG_B > 13.425 AND AMP11 <= 0.001717`, easily leading to the following transmuted query:

```
SELECT *
FROM EXOPL
WHERE MAG_B > 13.425 AND AMP11 <= 0.001717
```

It is worth noting that such an SQL query had very little chance of germinating in the initial stage of data exploration. This new query identified 22% of the initial positive examples, 0% of the negative examples and 1337 new tuples. These new tuples, representing stars around which the presence of revolving planets has not been studied, could thus be priority study targets due to their proximity, in the data exploration space, to a subset of stars around which planet presence has been confirmed.

The new SQL query turned out to be itself interesting: it showed the detectability limits of current instruments: for magnitudes greater than 13.425, i.e., for dimmer stars, it's mandatory to have star variability amplitudes less than 0.001717, i.e., the light emitted by the star must have a small variability.

Astrophysics scientists have found our approach satisfactory and easy to use, all of them having basic knowledge in SQL. The proposed transmuted query was a contribution per se. The use of machine learning techniques behind the scenes for proposing new SQL queries was completely transparent for them and very much appreciated. Clearly, other validations with domain experts should be conducted to assess the interest of our approach for data exploration in SQL, but these first results were very encouraging.

5. RELATED WORK

Data exploration is a very active research field in databases, data mining and machine learning. This section is organized according to several themes, namely query exploration, recommendation-based exploration, query by output and why-not queries.

Query exploration [25] shows that the time spent to assemble an SQL query is significantly higher than the query's execution time, even for SQL experts and decent-sized data. This is even more true in discovery-oriented applications, where the user does not know very well neither the database (schema / content), nor exactly what she is looking for (the right / exact queries to pose). [25] proposes a set of principles for a guided interaction paradigm, using the data to guide the query construction process. The user successively refines the query considering the results obtained at each iteration in order to reach a satisfactory solution.

[20] theorizes a new class of *exploration-driven applications*, characterized by *exploration sessions* with several interlinked queries, where the result of a query determines the formulation of the next query. [10] also talks about *interactive data exploration* applications characterized by *human-in-the-loop* analysis and exploration. It advocates the need for systems that provide session-oriented usage patterns, with sequences of related queries, where each query is the starting gate for the next one.

Query morphing [21] proposes a technique in which the user is presented with extra data via small changes of the initial query. By contrast, our solution exploits the examples and counter-examples sets obtained from the user's initial query, to obtain a new query via rules learned from these sets. We believe it is most of the time quite difficult to define *small* modifications for a given SQL query.

Some database exploration approaches are based on manual labeling of examples and counter-examples [34, 8, 30, 14, 23], on which machine learning approaches are eventually used to generate queries. Manual labeling has two main drawbacks: it is quickly tiring for the user, reducing system interactivity, simplicity and attractiveness; and it is far from trivial when the user does not know the data very well (frequently the case in an exploration task) or if the labeling criteria are complex (again, frequently the case, otherwise the user could probably obtain the data with a simple query).

Recommendation-based exploration [17] describes a template-based framework that recommends SQL queries as the user is typing in keywords. A top-k algorithm suggests queries derived from the queryable templates identified as relevant to the keywords provided by the user, based on a probabilistic model. The QueRIE framework [16] uses Web recommendation mechanisms to assist the user in formulating new queries. A query expressed by a user is compared with similar queries in the system log and a set of recommendations are proposed based on the behaviour of other users in a similar context. [6, 5] help the user formulate quantified, exploratory SQL queries. SnipSuggest [22] provides users with context-aware SQL query suggestions based on a log of historical queries. [15] assists the user in the exploratory task by suggesting additional *YMAL* items, not part of, but highly correlated with the results of an original query. It exploits offline computed statistics to identify subtuples appearing frequently in the original result, then builds exploratory queries, guiding the user to different directions in the database, not included in the original query. While we also make use of common statistics maintained by the optimizer, we incur no overhead by computing any other statistics. The entire process of aiding the user formulating her queries unfolds online.

Query by output solutions find a query that produces the data specified by the user, no more, no less. [31] finds an alternative path in the schema whose corresponding query produces the same data; an initial query may or may not be specified. The reformulated query in our approach does overlap the initial one to some extent, but is not equivalent to it. Similarly, [33] finds a join query, given its output, using arbitrary graphs. By contrast, [30] searches for a minimal valid project-join query starting from a few example tuples specified by the user. The generated query is expected to produce extra tuples, just like in our approach. The approach is however based on the user manually introducing example tuples. Likewise, [28] discovers queries whose answers include user-specified examples, for sample-driven schema mapping. [27] follows a similar approach to [30], but generates and ranks multiple queries that can partially contain the input tuples supplied by the user.

Why-not queries A database exploration approach tries to explain why a query on a database does not return desired tuples in the response, initially proposed in [11]. The framework proposed in [11] identifies the components from the guery evaluation plan responsible for filtering desired data items. This approach has been studied for different types of queries, as for reverse top-k queries [18]. As an alternative idea, the data provenance may be useful in our context by explaining why some tuples are included in the proposed exploration [9].

6. CONCLUSION AND DISCUSSION

In this paper we presented an approach that shows to be very promising in tackling one of the Big Data challenges: formulate SQL queries that correspond to what the analyst searches for and that efficiently execute on data of huge size. Starting from a query issued by a non-expert user, we explore the space of corresponding negation queries and we choose the one whose result size is as close as possible to the initial query's result size using a Knapsack-based heuristic. We then obtain a balanced set of examples and counter-examples that can feed a decision tree learning process. The learned model allows to directly rewrite the initial query using the obtained rules. This new transmuted query returns results that are similar to the initial query's ones, while also producing new tuples, which again are similar to the ones returned by the initial query. Such a form of diversity would have been practically impossible to achieve without the help of machine learning to formulate the query.

The user can also assess the global quality for the rewriting of her query, using diverse criteria that compare for instance the percentage of the new data obtained, or the number of tuples from the initial query that are retrieved. She can thus quickly evaluate the direction of her exploration, without having to first interpret the data she obtains for each query.

We have implemented and conducted several experiments to evaluate the main technical contribution of the paper, i.e., the Knapsack-based heuristic. Results are quite promising both in term of accuracy and performance. Preliminary validation results obtained on an astrophysics dataset with a simpler version of our current prototype [13], that automatically obtains a negation query from an initial so-called *discriminatory query*, are promising and open the way for an extensive experimental study.

The transparent integration of learning techniques with SQL is a significant change in scientists' way of working. The user just explores the data by posing questions in the golden SQL query language, both easy to use and intuitive. The user could therefore focus on the science part instead of the computational one. She does not have to switch between various systems and to re-load data several times, rending the exploration process ”seamless”.

A large number of possibilities has been opened up by this approach. We can easily imagine its extension to other types of SQL queries or to evaluate the scalability on very large datasets. We can also extend this work to pattern mining based on declarative languages like RQL [**?**], or, more generally, to all pattern types. For a given hypothesis (or pattern or query), the notion of examples and counter-examples seems to be relatively universal. This type of approach could thus play an interesting role in Big Data exploration in the years to come.

Acknowledgments: This work has been partially supported by `Petasky` and `QualiSky` projects under the CNRS `Mastodon` program. We thank the anonymous reviewers for their valuables remarks and Fabien Rouge and Christian Surace for their feedbacks on preliminary version of this work.

7. REFERENCES

[1] Accord .NET Framework. http://accord-framework.net/.

[2] Large Synoptic Survey Telescope. http://www.lsst.org//. [Online; accessed 11-Dec-2016].

[3] European Cooperation in the field of Scientific and Technical Research. Memorandum of understanding. http://w3.cost.eu/fileadmin/domain_files/TDP/Action_TD1403/mou/TD1403-e.pdf, 2014. [Online; accessed 11-Dec-2016].

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[5] 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.

[6] A. Abouzied, J. M. Hellerstein, and A. Silberschatz. Playful Query Specification with DataPlay. *PVLDB*, 5(12):1938–1941, Aug. 2012.

[7] R. Bellman. Notes on the theory of dynamic programming iv-maximization over discrete sets. *Naval Research Logistics Quarterly*, 3(1-2):67–70, 1956.

[8] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive Inference of Join Queries. In *EDBT*, pages 451–462, 2014.

[9] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330. Springer Berlin Heidelberg, 2001.

[10] U. Çetintemel, M. Cherniack, J. DeBrabant, Y. Diao, K. Dimitriadou, A. Kalinin, O. Papaemmanouil, and S. B. Zdonik. Query Steering for Interactive Data Exploration. In *CIDR*, 2013.

[11] A. Chapman and H. Jagadish. Why not? In *SIGMOD*, pages 523–534. ACM, 2009.

[12] B. Chardin, E. Coquery, M. Pailloux, and J. Petit. RQL: A SQL-Like Query Language for Discovering Meaningful Rules. In *ICDM*, pages 1203–1206, 2014.

[13] J. Cumin, J. Petit, F. Rouge, V. Scuturici, C. Surace, and S. Surdu. Requêtes discriminantes pour l'exploration des données. In *Extraction et Gestion des Connaissances*, pages 195–206, 2016.

[14] K. Dimitriadou, O. Papaemmanouil, and Y. Diao. Explore-by-example: an automatic query steering framework for interactive data exploration. In *SIGMOD*, 2014.

[15] M. Drosou and E. Pitoura. YmalDB: Exploring Relational Databases via Result-driven Recommendations. *The VLDB Journal*, 22(6):849–874, Dec. 2013.

[16] M. Eirinaki, S. Abraham, N. Polyzotis, and N. Shaikh. Querie: Collaborative database exploration. *IEEE TKDE*, 26(7):1778–1790, 2014.

[17] J. Fan, G. Li, and L. Zhou. Interactive SQL Query Suggestion: Making Databases User-friendly. In *ICDE*, ICDE '11, pages 351–362, 2011.

[18] Y. Gao, Q. Liu, G. Chen, B. Zheng, and L. Zhou. Answering why-not questions on reverse top-k queries. *PVLDB*, 8(7):738–749, 2015.

[19] 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.

[20] S. Idreos, O. Papaemmanouil, and S. Chaudhuri. Overview of Data Exploration Techniques. In *SIGMOD*, pages 277–281, 2015.

[21] 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):1474–1477, 2011.

[22] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. Snipsuggest: Context-aware autocompletion for SQL. *PVLDB*, 4(1):22–33, 2010.

[23] H. Li, C. Chan, and D. Maier. Query From Examples: An Iterative, Data-Driven Approach to Query Construction. *PVLDB*, 8(13):2158–2169, 2015.

[24] A. Marchetti-Spaccamela and G. Romano. On different approximation criteria for subset product problems. *Information processing letters*, 21(4):213–218, 1985.

[25] A. Nandi and H. V. Jagadish. Guided Interaction: Rethinking the Query-Result Paradigm. *PVLDB*, 4(12):1466–1469, 2011.

[26] C. T. Ng, M. S. Barketau, T. C. E. Cheng, and M. Y. Kovalyov. "Product Partition" and related problems of scheduling and systems reliability: Computational complexity and approximation. *European Journal of Operational Research*, 207(2):601–604, 2010.

[27] F. Psallidas, B. Ding, K. Chakrabarti, and S. Chaudhuri. S4: Top-k Spreadsheet-Style Search for Query Discovery. In *SIGMOD*, pages 2001–2016, 2015.

[28] L. Qian, M. J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *SIGMOD*, pages 73–84, 2012.

[29] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[30] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. In *SIGMOD*, pages 493–504, 2014.

[31] Q. T. Tran, C. Chan, and S. Parthasarathy. Query by output. In *SIGMOD*, pages 535–548, 2009.

[32] D. Victor. The Ashley Madison Data Dump, Explained. New York Times, 2015-08-20, 2015.

[33] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse Engineering Complex Join Queries. In *SIGMOD*, SIGMOD '13, pages 809–820, New York, NY, USA, 2013. ACM.

[34] M. M. Zloof. Query-by-example: A Data Base Language. *IBM Syst. J.*, 16(4):324–343, Dec. 1977.

Task-Optimized Group Search for Social Internet of Things

Chih-Ya Shen[⋆], Hong-Han Shuai[†], Kuo-Feng Hsu[‡], Ming-Syan Chen[‡]
[⋆]National Tsing Hua University, Hsinchu, Taiwan
[†]National Chiao Tung University, Hsinchu, Taiwan
[‡]National Taiwan University, Taipei, Taiwan
chihya@cs.nthu.edu.tw, hhshuai@nctu.edu.tw, {r03921038,mschen}@ntu.edu.tw

ABSTRACT

With the maturity and popularity of Internet of Things (IoT), the notion of Social Internet of Things (SIoT) has been proposed to support novel applications and networking services for the IoT in more effective and efficient ways. Although there are many works for SIoT, they focus on designing the architectures and protocols for SIoT under the specific schemes. How to efficiently utilize the collaboration capability of SIoT to complete complex tasks remains unexplored. Therefore, we propose a new query, namely *Task-Optimized Group Search (TOGS)*, to address this need. TOGS aims to extract the target SIoT group such that the target SIoT group will be able to easily communicate with each other while maximizing the accuracy of performing the given tasks. We propose two problem formulations, namely *Bounded Communication-loss TOSS (BC-TOSS)* and *Robustness Guaranteed TOSS (RG-TOSS)*, for different communication scenarios, and prove that they are both NP-Hard and inapproximable. We propose a polynomial-time algorithm with performance bound for BC-TOSS, and an efficient polynomial-time algorithm to obtain good solutions for RG-TOSS. The experimental results on real datasets indicate that our proposed algorithms outperform other baselines.

1. INTRODUCTION

With the maturity and popularity of Internet of Things (IoT), it has been widely recognized that the Internet of Things (IoT) is the next paradigm shift. The future Internet will embody a tremendous number of objects that provide valuable information and controllable actions. Moreover, with the capability of interactions among each other, objects can collaborate with other counterparts toward providing services to the end users, e.g., environmental monitoring, surveillance, smart home, health care, and product management.

Recently, since it has been shown that a large number of users tied in a social network can provide far more accurate answers to complex problems than a single user [18], a recent line of studies investigates the opportunities of integrating social networking concepts into solving complex problems. Several schemes have been proposed to exploit social networks for question answering via crowdsourcing [1], P2P routing [3], or web security [22]. Meanwhile, by incorporating the concept of social network into IoT, the idea of Social Internet of Things (SIoT) has been proposed to support novel applications and networking services for the IoT in more effective and efficient ways.

However, current research focuses on designing the architectures and protocols for SIoT under the specific schemes. For example, Kosmatos et al. integrated the RFID and smart object-based infrastructures towards building blocks of SIoT [8]. Nitti et al. proposed two trustworthiness management models to suggest strategies of establishing trustworthiness among nodes to isolate malicious nodes [12]. Moreover, to build reliable communication for SIoT, Chen et al. proposed an adaptive trust management protocol which adaptively chooses the best trust parameter settings w.r.t. the changing IoT social conditions to assess the trust correctly and maximize the application performance [2]. To the best of our knowledge, how to efficiently utilize the collaboration capability of SIoT to complete the complex tasks remains unexplored.

To complete the complex tasks under SIoT environments, one basic solution is to specify all the required functions of the complex task and perform the required functions on the corresponding SIoT. However, since the number of SIoT objects with the same required functions is tremendous, it is extremely redundant and inefficient to perform the functions on all the compliant SIoT objects. Meanwhile, users also pay the usage cost based on the amount of utilization (pay as you go) in the forms of rental fee or the cost of requested data. Therefore, we adopt the semantic of top-k query to search the optimal group of SIoT objects with the largest success possibility for completing the complex task. Moreover, due to the network reliability of SIoT, it is desirable that each component within the selected group is tightly-coupled or at least not far from each other.

Take Figure 1 as an example. Since the number of catastrophic wildfires has been steadily rising, the government plans to build a wildfire alarm system from the existing SIoT objects. The wildfire alarm prediction task is correlated to accumulative rainfall, temperature, wind speed, and accumulative snowfall according to previous study [6], and each SIoT object can report at least one measurement within an

©2017, Copyright is with the authors. Published in Proc. 20th International Conference on Extending Database Technology (EDBT), March 21-24, 2017 - Venice, Italy: ISBN 978-3-89318-073-8, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

Series ISSN: 2367-2005 10.5441/002/edbt.2017.11

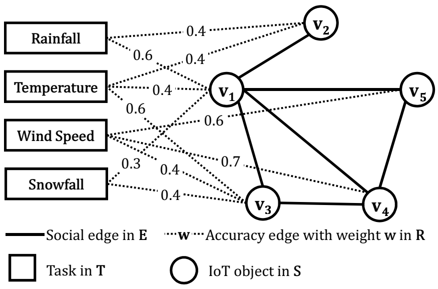


Figure 1: Illustrative example for wildfire detection

accuracy threshold.[1] Therefore, the alarm system issues a top-k query on SIoT and finds the group that maximizes the accuracy of all related measurement. Moreover, under the SIoT environment, each SIoT object replicates its measurement data to its trustworthy "friends" for reliability, fault-tolerance, or accessibility. Therefore, for the data reliability, it is desirable that each component in the selected group is within h-hop from each other or has at least some friends.

Specifically, we propose a new framework, namely, *Task-Optimized Group Search (TOGS)*, to search the best group under the abovementioned SIoT environments. Given a heterogeneous social graph, the set of tasks, the social relationships between SIoT objects, and the relationships between each SIoT object and the task, we propose a new problem family, namely, *Task-Optimized SIoT Selection (TOSS)*, to find the best group of IoT objects for a given set of tasks in the task pool.

To consider different application needs, we propose two different problem formulations for TOSS, namely *Bounded Communication-loss TOSS (BC-TOSS)* and *Robustness Guaranteed TOSS (RG-TOSS)*. While the objective of the two problems are both maximizing the accuracy of performing the given tasks, BC-TOSS aims to bound the communication loss between different SIoT objects, and the goal of RG-TOSS is to provide robustness for message transmission among different SIoT objects. We formulate the problems and prove that they are both NP-Hard and inapproximable within any factor. We propose an error-bounded algorithm with guaranteed performance, namely *Hop-bounded Accuracy-optimized SIoT Extraction (HAE)*, to obtain in polynomial time a solution with objective value no worse than the optimal solution with a bounded error for BC-TOSS. For RG-TOSS, we propose an efficient algorithm to obtain good solutions in polynomial time, namely *Robustness-Aware SIoT Selection (RASS)* which includes effective processing strategies such as *Core-based Robustness Pruning*, *Accuracy-Optimization Pruning*, *Robustness-Guaranteed Pruning*, and *Accuracy-oriented Robustness-aware Ordering*. We conduct a user study for evaluating the effectiveness of the problem formulations, and perform extensive experiments on real datasets to evaluate the proposed algorithms. Experimental results show that our proposed algorithms significantly outperform other baselines. The contributions are summarized as follows.

- For completing complex task in SIoT environment, we propose to model the social edges within SIoT objects with accuracy edges in a heterogeneous graph and propose two different problem formulations, i.e., *BC-TOSS* and *RG-TOSS*, to find suitable SIoT objects. To our best knowledge, there is no real system or existing work in the literature that addresses the issue of group search in SIoT environment.
- We prove that both formulations are NP-Hard and inapproximable within any factors. We then propose a polynomial-time algorithm with performance guarantee and bounded error, namely *Hop-bounded Accuracy-optimized SIoT Extraction (HAE)* for the BC-TOSS problem. We also propose an effective polynomial time algorithm, namely *Robustness-Aware SIoT Selection (RASS)*, to find good solutions for the RG-TOSS problem.
- We conduct a user study on 100 users to validate our two problem formulations. Moreover, we perform extensive experiments on two real datasets. The results show that the proposed algorithms outperform the baselines in terms of objective values and efficiency.

The rest of this paper is organized as follows. Section 2 introduces the related works. Section 3 formulates the problems and proves that the proposed problems are NP-Hard and inapproximable within any factor. Sections 4 and 5 propose algorithms to BC-TOSS and RG-TOSS problems, respectively. Section 6 shows the experimental results and Section 7 concludes this paper.

2. RELATED WORK

A recent line of SIoT research focuses on designing the architectures and protocols for facilitating SIoT under the specific schemes [8, 12, 2]. For example, Nitti et al. propose two trustworthiness management models to suggest strategies of establishing trustworthiness among nodes so that malicious nodes are isolated [12]. Yao et al. propose a joint probabilistic framework for fusing the social relationships between users and IoT objects to improve the accuracy of IoT recommendations [21]. However, these works do not take the capability of collaboration between SIoT into consideration. Moreover, our goal is to find the optimal group of SIoT to accomplish certain tasks, not recommending a single IoT.

To find a cohesive group, many different measurements have been reported in the literature, e.g., diameter [19], density [4, 5], clique and its variations [11]. However, the above works only consider the characteristics inside the group on the existing friendship edges, but *TOGS* needs to consider the accuracy of the assigned task and the "social" tightness among IoT objects. Therefore, new algorithms are necessary to take both types of edges into account. Researches have been proposed to find a socially close group of individuals to invite for activities. In [17], given a query group, the total degree of the community containing this group is maximized. The spatial factor is considered in [10, 20, 23]. Furthermore, the willingness to participate activities is considered in [16]. All the above works keep the social tightness of the target group on the friendship network while maximizing or maintaining some characteristic people care about in an activity. In contrast, in this paper, the accuracy is considered to be maximized so that the query task can find a proper target SIoT group to complete the complex tasks.

On the other hand, expert team formation has attracted a lot of research interests. Forming an expert team is to find

[1] The SIoT objects that cannot report any related measurement can be filtered at the beginning.

Table 1: Notation Summary.

Symbol	Meaning
Q	Query group
$d_S^E(F)$	Largest shortest path distance in F on E
$I_F(t)$	Incident weight of $t \in T$
$deg_H^E(v)$	Inner degree of v in H
$\Omega(F)$	Objective value of target group F
$\alpha(u)$	Sum of incident accuracy edge weight of $u \in S$

a set of experts the required skills, while the communication cost among the chosen experts is minimized so that team members can communicate with each other efficiently. Several communication costs have been proposed under different considerations. For example, Lappas et al. find a team that covers the required skills and minimizes the social diameter of the team or the total edge weight of the spanning tree within the team [9]. Moreover, projects usually require a leader for guiding the direction and negotiation among members. Therefore, Kargar et al. [7] proposed to select a leader for each skill and minimize the social distance from the skill members to each skill leader. In [15], the authors further consider both spatial proximity and skill requirements for finding quick response teams. In contrast, our paper is the first to study the task-optimized group search problem considering SIoT as the input. Based on both social edges and accuracy edges, *TOGS* aims to find the target group with enough social tightness on SIoT to ensure the reliability and while maximizing the accuracy to query tasks.

3. PROBLEM FORMULATIONS

In this paper, we consider a family of *Task-Optimized SIoT Selection (TOSS)* problems on a heterogeneous graph which aim to find the best group of SIoT objects for certain tasks by considering the interactions among different SIoT objects. Specifically, given the heterogeneous graph $G = (T, S, E, R)$, the vertex set T is the *task pool*, i.e., the union of the tasks the SIoT objects can achieve such as measuring humidity, rainfall. Vertex set S represents the set of SIoT objects, where the social relationships among them are captured by the unweighted *social edge* set E, $S \times S \to E$. Here, a social edge $(u, v) \in E$ represents that SIoT objects u and v can communicate (e.g., using the same communication protocol or equipping the same transmission hardware). For the ease of presentation, we use the term *SIoT Graph*, $G_S = (S, E)$, to describe the graph composed of the SIoT objects set S and the corresponding social edge set E. Finally, R is the set of *task accuracy edge* (*accuracy edge* for short), where each accuracy edge $r = [t, v]$ linking a task vertex $t \in T$ and an SIoT vertex $v \in S$ indicates the accuracy for the SIoT object v to perform task t as the edge weight $w[t, v] \in (0, 1]$. An illustrative example of the heterogeneous graph $G = (T, S, E, R)$ is shown in Figure 1. Table 1 summarizes the notations.

Given the heterogeneous graph $G = (T, S, E, R)$ mentioned above, a query group $Q \subseteq T$ and the desired size p of SIoT objects, the goal of the TOSS problems is to find the group $F \subseteq S$ of exactly p SIoT objects to optimize the accuracy (the definition of *optimizing the accuracy* will be detailed later) of the selected tasks in Q. The size constraint p here represents a budget constraint, i.e., how many SIoT objects we plan to control or carry according to our application scenario. Moreover, based on different practical needs, we apply different constraints on Q to either reduce the communication loss or to increase the robustness of the selected SIoT objects in F. Based on the different constraints, we propose two different problem formulations and algorithm designs.

Specifically, we first propose the *Bounded Communication-loss TOSS (BC-TOSS)* problem by taking into account the communication loss between different SIoT objects in addition to optimizing the accuracy of the selected tasks in Q. To achieve this, we place an upper bound on the hop distance between each pair of SIoT objects in order to limit the number of message forwarding. In other words, the constraint of BC-TOSS is to require the *hop distance* between each pair of vertices in F on E to be at most h, i.e., $d_S^E(F) \leq h$, to reduce the potential communication loss. Please note that since an SIoT object u can forward messages even if it is not selected in F, therefore, the shortest path considered by $d_S^E(F)$ can go through vertices in S but outside F. For example, in Figure 1, if $F = \{v_2, v_3\}$, $d_S^E(F) = 2$ because the shortest path can go through $v_1 \notin F$.

In the second problem, namely *Robustness Guaranteed TOSS (RG-TOSS)*, we pay special attention to the number of different message transmission paths. In other words, RG-TOSS, in addition to optimizing the accuracy of the selected tasks in Q, also requires that each SIoT object in F has at least k neighboring SIoT objects for successfully transmitting the messages. That is, each vertex $v \in F$ must have at least k neighboring vertices also in F.

To measure the solution quality of the returned group F, we consider the sum of the accuracy edge weights incident to each vertex t in Q. Let $I_F(t)$ denote the sum of incident accuracy edge weights of $t \in Q$ to the target group F (*incident weight* of t for short), i.e., $I_F(t) = \sum_{v \in F} w[t, v]$. We then use the sum of incident weights over all tasks t in the task group Q to F to represent the aggregated quality of the returned group F corresponding to Q. In other words, the objective function of the returned group F is defined as $\Omega(F) = \sum_{t \in Q} I_F(t)$. In this paper, we aim to maximize the objective function $\Omega(F)$ to ensure that the tasks in Q are most likely to succeed. Furthermore, we also include an *accuracy constraint* τ in the problem formulation. This accuracy constraint requires that the edge weight of each accuracy edge between Q and F must be at least τ, to ensure the worst case performance of the returned target group.

In the following, we formally formulate the two TOSS problems, namely *Bounded Communication-loss TOSS (BC-TOSS)* and *Robustness Guaranteed TOSS (RG-TOSS)*. We also prove that the proposed two TOSS problems are both NP-Hard and inapproximable within any factors unless P=NP.

3.1 Bounded Communication-loss TOSS (BC-TOSS)

The Bounded Communication-loss TOSS (BC-TOSS) problem is defined as follows.

Problem: Bounded Communication-loss TOSS (BC-TOSS).
Given: Heterogeneous graph $G = (T, S, E, R)$, query group $Q \subseteq T$, hop constraint $h \geq 1$, size constraint $p > 1$, and accuracy constraint $\tau \in [0, 1]$
Objeective: To find a target group $F \subseteq S$ where i) $|F| = p$, ii) $d_S^E(F) \leq h$, and iii) $w[u, v] \geq \tau, \forall u \in Q, v \in F, [u, v] \in R$, such that $\Omega(F) = \sum_{t \in Q} I_F(t)$ is maximized.

Solving the proposed BC-TOSS problem is very challenging due to the interplay of two different edge sets, i.e., groups with largest objective value may not satisfy the hop constraint. In the following, we first analyze the hardness of BC-TOSS by proving that the BC-TOSS problem is NP-Hard. In addition, we prove that there exists no polynomial time approximation algorithm for BC-TOSS. In other words, BC-TOSS is inapproximable within any factor.

THEOREM 1. *BC-TOSS is NP-Hard and inapproximable within any factor.*

PROOF. We prove that BC-TOSS is an NP-Hard problem with the reduction from the $\widehat{p}$-clique problem, which is an NP-Complete problem. Given a graph $G_c = (V_c, E_c)$, where V_c is the set of vertices and E_c is the set of undirected and unweighted edges, and an integer $\widehat{p}$, the decision problem of $\widehat{p}$-clique is to answer whether there exists a subgraph $C_c \subseteq G_c$ such that i) C_c has exactly $\widehat{p}$ vertices, i.e., $|C_c| = \widehat{p}$, and ii) C_c is a complete graph, i.e., $d^E_S(C_c) = 1$, where $d^E_S(C_c)$ is the longest shortest path length on vertex set S and edge set E among all the vertex pairs in C_c.

We transform each instance of $\widehat{p}$-clique to an instance of BC-TOSS as follows. We construct the input graph of BC-TOSS $G = (T, S, E, R)$ by letting $S = V_c$, $E = E_c$, while the task pool T, the accuracy edges in R, the corresponding edge weights, and the query group Q are set arbitrarily. The parameters of BC-TOSS are set as $p = \widehat{p}$, $h = 1$, and $\tau = 0$. In the following, we prove that the decision problem $\widehat{p}$-clique returns TRUE if and only if BC-TOSS has a feasible solution. We first prove the sufficient condition. If $\widehat{p}$-clique returns TRUE with a solution C_c with $d^E_S(C_c) = 1$ and $|C_c| = \widehat{p}$, then C_c must be a feasible solution to BC-TOSS because $d^E_S(C_c) \leq h = 1$ and $|C_c| = \widehat{p} = p$. We then prove the necessary condition. If F is a feasible solution to BC-TOSS, then $d^E_S(F) \leq h = 1$ and $|F| = p$ must hold, which implies that F is also a complete graph of size $\widehat{p} = p$.[2] Therefore, F is also a solution to $\widehat{p}$-clique. This proves that BC-TOSS is NP-Hard.

Finally, we prove hat there exists no approximation algorithm for BC-TOSS unless P=NP. Note that BC-TOSS will return $\Omega(F) = 0$ if $F = \varnothing$, i.e., no feasible solution exists for BC-TOSS. Therefore, if BC-TOSS has a polynomial-time approximation algorithm with an arbitrarily large ratio $\delta < \infty$, the above proof indicates that i) the algorithm is able to obtain a feasible solution to BC-TOSS if $\widehat{p}$-clique returns TRUE, and ii) any BC-TOSS instance with the algorithm returning a feasible solution implies that the corresponding instance in $\widehat{p}$-clique is TRUE. That is, the δ-approximation algorithm can solve $\widehat{p}$-clique in polynomial time, implying that P=NP. Therefore, BC-TOSS has no polynomial-time approximation algorithm unless P=NP. □

Theorem 1 states that the BC-TOSS problem is NP-Hard and inapproximable within any factor. However, we observe that if we slightly relax one constraint of the BC-TOSS problem, we are able to obtain the solution no worse than the optimal solution within polynomial time. We detail this algorithm with performance bound in Section 4.

3.2 Robustness Guaranteed TOSS (RG-TOSS)

To find a target group to ensure the communication robustness, i.e., each SIoT object in the target group is able to transmit or backup its data through a number of different neighboring objects, one promising way is to ensure the minimum number of neighbors each SIoT object has in the target group. This motivates us to introduce the degree constraint to guarantee the robustness of communications. We first denote the *inner degree* of a vertex v according to the edge set E in a subgraph $H \subseteq G$ as $deg^E_H(v)$, which is the number of vertices $u \in H$ such that (u, v) is an edge in E. Then, we formally formulate the Robustness Guaranteed TOSS problem as follows, which incorporates the degree constraint in the target group and has the identical objective function and size constraint as the BC-TOSS problem.

Problem: Robustness Guaranteed TOSS (RG-TOSS).
Given: Heterogeneous graph $G = (T, S, E, R)$, query group $Q \subseteq T$, degree constraint $k \geq 1$, size constraint $p > 1$, and accuracy constraint $\tau \in [0, 1]$.
Objeective: To find a target group $F \subseteq S$ where i) $|F| = p$, ii) $deg^E_F(v) \geq k, \forall v \in F$, iii) $w[u, v] \geq \tau, \forall u \in Q, v \in F, [u, v] \in R$, such that $\Omega(F) = \sum_{u \in Q} I_F(u)$ is maximized.

Similar to BC-TOSS, the interplay of the constraints and objective functions on social edges and accuracy edges makes processing RG-TOSS very challenging, especially when the degree constraint of RG-TOSS requires each SIoT object in the returned group to have at least k neighbors in the same group. Please note that this degree constraint requires the *inner degree* to be at least k, which models a more practical situation that we only have control on the selected SIoT objects, i.e., we do not replicate data to or communicate with SIoT objects outside the selected group. Intuitive approaches such as greedily choosing vertices to optimize the objective value does not work because it does not consider the degree constraint and may not obtain feasible solutions. In fact, RG-TOSS is also NP-Hard and inapproximable within any factor, which is proved as follows.

THEOREM 2. *RG-TOSS is NP-Hard and inapproximable within any factors unless P=NP.*

PROOF. We prove that RG-TOSS is NP-Hard with the reduction from an NP-Complete problem, namely $\tilde{k}$-plex problem[14]. Given graph $G = (\tilde{V}, \tilde{E})$ and positive integers $\tilde{p}$ and $\tilde{k}$, the decision problem $\tilde{k}$-plex determines if there exists a set of vertices $C \subseteq \tilde{V}$, such that $deg^{\tilde{E}}_{\tilde{C}}(u) \geq |C| - \tilde{k}, \forall u \in C$ and $|C| = \tilde{p}$.

We transform an instance of the $\tilde{k}$-plex problem to an instance of the RG-TOSS instance by first creating the heterogeneous graph $G = (T, S, E, R)$ with $S = \tilde{V}$ and $E = \tilde{E}$. The task pool T, the set of accuracy edges, and the corresponding accuracy edge weights are set arbitrarily. Moreover, the query group $Q \subseteq T$ is also chosen arbitrarily and $k = \tilde{p} - \tilde{k}$, $p = \tilde{p}$, $\tau = 0$.

We first prove the sufficient condition. If there exists a set $C \subseteq \tilde{V}$ with $deg^{\tilde{E}}_{\tilde{C}}(u) \geq |C| - \tilde{k}, \forall u \in C$ and $|C| = \tilde{p}$, then C must be a feasible solution to the RG-TOSS instance. For the necessary condition, if F is a feasible solution to RG-TOSS, then $|F| = p$ and $deg^{\tilde{E}}_{\tilde{C}}(u) \geq |C| - \tilde{k}, \forall u \in C$ must hold. Therefore, F is also a $\tilde{k}$-plex. This proves that RG-TOSS is NP-Hard.

For the inapproximability, if there exists any δ-approximation

[2] Please note that when $d^E_S(F) = 0$, F contains at most 1 vertex, not satisfying the requirement of BC-TOSS which asks $p > 1$.

algorithm for any $\delta < \infty$, then such δ-approximation algorithm must be able to obtain a feasible solution of TG-TOSS in polynomial time, which is equivalent to solving the NP-Complete problem $\tilde{k}$-plex in polynomial time. Therefore, RG-TOSS is inapproximable within any factor unless P=NP. The theorem follows. □

Since there exists no approximation algorithm for the RG-TOSS problem unless P=NP, we propose an effective and efficient polynomial-time algorithm to tackle the challenges brought by the interplay of RG-TOSS. We detail the algorithm design in Section 5.

4. ALGORITHM FOR BC-TOSS WITH PERFORMANE GUARANTEE

Theorem 1 in Section 3.1 states that BC-TOSS is NP-Hard and inapproximable within any factor. One simple approach is to enumerate all the combinations to find the optimal solution of BC-TOSS. Due to the large search space, the time complexity of such intuitive approach would be $O(|V|^p)$ which makes it computationally expensive and inapplicable for a large-scale Social IoT network. However, we observe that if we slightly relax the hop constraint, it is possible to find a polynomial time algorithm that can find a solution no worse than the optimal solution (performance guarantee) to BC-TOSS with a *bounded error*. Therefore, in this section, we propose a polynomial time algorithm, namely *Hop-bounded Accuracy-optimized SIoT Extraction (HAE)*, to find the solution with the objective value no worse than the optimal solution while the distance between each pair of vertices on E in the returned group may exceed h, but is guaranteed to be within $2h$. We formally prove the performance guarantee and the error bound of the proposed algorithm.

To avoid generating infeasible solutions, the proposed HAE algorithm first performs a preprocessing step to guarantee that each SIoT object in S has all its incident accuracy edge weights at least τ. That is, this preprocessing step removes each vertex $u \in S$ with an accuracy edge $[u, v]$ for some $v \in Q$ with $w[u, v] < \tau$. Then, the vertices in S which have no incident accuracy edge are also removed because including them in the solution will not increase the objective value.

Afterwards, the HAE algorithm performs a *Sieve Step* to filter out redundant vertices. If an SIoT object $v \in S$ is in the returned group F, then any vertex $u \in F$ must satisfy the following inequality: $d_S^E(u, v) \leq h$. Therefore, the Sieve Step first constructs the candidate set S_v for each SIoT object $v \in S$ where S_v contains only the vertices that are able to form the target group with v, i.e., S_v contains only the vertices within h hops on E from v. Take Figure 1 as an example. Assume $Q =$ {Rainfall, Temperature, Wind Speed, Snowfall}, $p = 3$, $h = 1$, and $\tau = 0.25$. In the Sieve Step, for example, $S_{v_1} = \{v_1, v_2, v_3, v_4, v_5\}$ because all these SIoT objects are within $h = 1$ hop from v_1, and $S_{v_3} = \{v_1, v_3, v_4\}$.

After the Sieve Step is complete, algorithm HAE performs the *Refine Step* to examine each vertex in S_v. Specifically, given SIoT object $u \in S$, we denote $\alpha(u)$ the sum of accuracy edge weights linking from u to the tasks in Q, i.e., $\alpha(u) = \sum_{s \in Q} w[u, s]$. Then, to maximize the objective function, the Refine Step selects p vertices from S_v which have the maximum $\alpha(u)$ and constructs a candidate solution $\mathbb{S}_v$ for v. If $\Omega(\mathbb{S}_v)$ is larger than that of the currently best solution $\mathbb{S}^*$, Algorithm HAE updates $\mathbb{S}^*$ as $\mathbb{S}_v$. Algorithm HAE repeats the examinations of $v \in S$ to construct different candidate solutions, and returns the solution with the maximum objective value as the target group F. Return to our running example in Figure 1. After this step, $\mathbb{S}_{v_1} = \{v_1, v_2, v_3\}$, and $\mathbb{S}_{v_4} = \{v_1, v_3, v_4\}$. Please note that S_{v_2} does not need to be examined because $|S_{v_2}| = 2 < p$, i.e., no feasible solution can be constructed from S_{v_2}. After examining all S_v, the returned target group $F = \{v_1, v_2, v_3\}$, which is the optimal solution.

One major weakness of the steps mentioned above is that algorithm HAE needs to scan over all vertices in S to construct candidate solutions and extract the best one among them. However, this may incur large computation overhead. We observe that if we examine each vertex $v \in S$ in some predefined order, then some $v \in S$ does not need to be examined because the vertices in the corresponding S_v cannot generate a solution better than the best solution obtained so far. Therefore, we propose a vertex-visiting ordering and lookup strategy, namely *Incident Weight Ordering with Top-p Objects Lookup (ITL)* and a powerful pruning strategy, called *Accuracy Pruning (AP)*, to avoid unnecessary search space exploration. ITL visits each vertex $v \in S$ in descending order of $\alpha(v)$, which enables Accuracy Pruning to better estimate the solution quality in each S_v to avoid redundant examinations. Moreover, ITL enables quick candidate solution $\mathbb{S}_v$ generation without sorting the vertices in S_v to extract the top-p vertices with the maximum $\alpha(\cdot)$ values.

Specifically, we associate with each vertex $v \in S$ a list L_v, which is used to store the top-p vertices of the maximum $\alpha(\cdot)$ in S_v. Each time when algorithm HAE examines vertex v and constructs the corresponding S_v in the descending order of $\alpha(v)$, HAE inserts v into each vertex u's list $L_u, \forall u \in S_v$ if $|L_u| < p$. For example in Figure 1, v_3 is visited first because $\alpha(v_3)$ is the largest. After constructing $S_{v_3} = \{v_1, v_3, v_4\}$, HAE also inserts v_3 into L_{v_1}, L_{v_3}, L_{v_4}. The following Lemma 1 proves that the above-mentioned strategy can guarantee that L_u always stores the top-$|L_u|$ vertices with the maximum $\alpha(\cdot)$ in L_u.

LEMMA 1. *For any vertex $u \in S$, its associated L_u stores the top-$|L_u|$ vertices with the maximum $\alpha(\cdot)$ in S_u. Moreover, if $|L_u| < p$, then $\alpha(x) \leq \alpha(u), \forall x \in S_u \backslash L_u$.*

PROOF. HAE visits the vertices $v \in S$ in descending order of $\alpha(v)$. Therefore, for any vertex u, the vertices in its list L_u must be visited before u, leading to $\alpha(x) \geq \alpha(u), \forall x \in L_u$. Moreover, if $u \in S_v$, then $v \in S_u$ as well. Therefore, the vertices in L_u must be in S_u. Since L_u stores at most the first p vertices visited by HAE in S_u, L_u stores the top-$|L_u|$ vertices with the maximum $\alpha(\cdot)$ in S_u.

Please note that the vertices in L_u must have been visited by HAE and $\alpha(y) \geq \alpha(u), \forall y \in L_u$. If $|L_u| < p$, then the vertices in $S_u \backslash L_u$ must not have been visited by HAE yet. According to the vertex-visiting ordering, $\alpha(x) \leq \alpha(u), \forall x \in S_u \backslash L_u$ holds. The lemma follows. □

HAE is equipped with a powerful pruning, namely *Accuracy Pruning*, which can avoid the examination of redundant S_v which never generates better solutions than the currently best solution $\mathbb{S}^*$. Accuracy Pruning works as follows. When Algorithm HAE visits a vertex $v \in S$, before constructing S_v to include the vertices within h hops of v, it first examines the list L_v to check if S_v has a chance to generate a better solution than the currently best solution $\mathbb{S}^*$. If S_v cannot, HAE skips v and proceeds to examine the next vertex. This saves the computation of traversing the graph to construct S_v. Specifically, the following lemma shows the pruning condition and the correctness of the Accuracy Pruning.

LEMMA 2. ***Accuracy Pruning.*** *Given $v \in S$ and the currently best solution $\mathbb{S}^*$, if $\Omega(L_v) + (p - |L_v|)\alpha(v) \leq \Omega(\mathbb{S}^*)$ holds, S_v can be safely pruned without examination.*

PROOF. Let M_v denote the p vertices with the maximum $\alpha(\cdot)$ values in S_v, and $\widehat{S}_v$ be an arbitrary subset of S_v with $|\widehat{S}_v| = p$. Then $\Omega(M_v) \geq \Omega(\widehat{S}_v)$ must hold. We would like to show that if S_v is pruned by Accuracy Pruning, then there does not exist any $\widehat{S}_v$ such that $\Omega(\widehat{S}_v) > \Omega(\mathbb{S}^*)$.

We prove by contradiction. Assume that $\Omega(M_v) > \Omega(\mathbb{S}^*)$, then $\Omega(M_v) > \Omega(\mathbb{S}^*) \geq \Omega(L_v) + (p - |L_v|)\alpha(v)$ must hold because S_v is pruned by Accuracy Pruning. Case 1) If $|L_v| = p$, then $\Omega(M_v) = \Omega(L_v)$ according to Lemma 1, and we will conclude that $\Omega(M_v) > \Omega(M_v)$ which leads to a contradiction. 2) If $|L_v| < p$, $\sum_{x \in M_v} \alpha(x) > \sum_{x \in L_v} \alpha(x) + (p - |L_v|)\alpha(v)$ holds. According to Lemma 1, $\sum_{x \in (M_v \backslash L_v)} \alpha(x) > (p - |L_v|)$ holds. Therefore, there exists $x \in (M_v \backslash L_v) \subseteq S_v$ L_v such that $\alpha(x) > \alpha(v)$, which contradicts with Lemma 1. Since the above two cases lead to contradictions, $\Omega(M_v) \leq \Omega(\mathbb{S}^*)$ must hold, which leads to $\Omega(\mathbb{S}^*) \geq \Omega(M_v) \geq \Omega(\widehat{S}_v)$. Therefore, if S_v is pruned by Accuracy Pruning, any p-vertex subset $\widehat{S}_v \subseteq S_v$ must have $\Omega(\widehat{S}_v) \leq \Omega(\mathbb{S}^*)$, i.e., S_v cannot generate any solution with objective value better than the currently best solution $\mathbb{S}^*$. The lemma follows. □

Return to our running example in Figure 1. When Algorithm HAE visits v_4, $L_{v_4} = \{v_1, v_3\}$, and the currently best solution $\mathbb{S}^*$ is $\{v_1, v_2, v_3\}$ with $\Omega(\mathbb{S}^*) = 3.5$. In this case, $\Omega(L_{v_4}) + (p - |L_{v_4}|) \cdot \alpha(v_4) = 2.7 + 1 \cdot 0.7 = 3.4 < \Omega(\mathbb{S}^*) = 3.5$, and Accuracy Pruning prunes v_4. Therefore, Algorithm HAE avoids examining v_4 and does not need to construct S_{v_4} because any subset with p vertices of S_{v_4} will never become a solution better than $\mathbb{S}^*$. The pseudo code of algorithm HAE is shown in Algorithm 1.

In the following, we prove the performance guarantee and error bound of the proposed algorithm. We first prove that, if the optimal solution $\mathbb{S}^{OPT}$ contains a vertex $v \in S$, then $\mathbb{S}^{OPT} \subseteq S_v$ must hold. That is, algorithm HAE does not need to examine any vertex outside S_v if $v \in \mathbb{S}^{OPT}$.

LEMMA 3. *If the optimal solution $\mathbb{S}^{OPT}$ contains vertex $v \in S$, then $\mathbb{S}^{OPT} \subseteq S_v$ holds.*

PROOF. Assume that there exists vertex $v' \in \mathbb{S}^{OPT}$ such that $\mathbb{S}^{OPT}$ is not a subset of $S_{v'}$. Since $\mathbb{S}^{OPT}$ is not a subset of $S_{v'}$, we can find a vertex $u' \in \mathbb{S}^{OPT}$ such that $u' \notin S_{v'}$. In other words, $d^E_S(u', v') > h$ where $d^E_S(u', v') > h$ is the shortest path distance from u' to v' on E. However, from the hop constraint, we know that $d^E_S(u, v') \leq h, \forall u \in \mathbb{S}^{OPT}$ which is a contradiction. Therefore, if $\mathbb{S}^{OPT}$ contains vertex $v \in S$, then $\mathbb{S}^{OPT} \subseteq S_v$ holds. □

We now prove that the proposed HAE algorithm is able to obtain the solution no worse than the optimal solution (performance guarantee) with an error bound h.

THEOREM 3. *The solution F returned by algorithm HAE is no worse than the optimal solution $\mathbb{S}^{OPT}$ to BC-TOSS with an error bound h. That is, $\Omega(F) \geq \Omega(\mathbb{S}^{OPT})$ with $d^E_S(F) \leq 2h$.*

Algorithm 1: Hop-bounded Accuracy-optimized SIoT Extraction (HAE)

```
Input: $G = (T, S, E, R)$, $Q$, $h$, $p$, $\tau$
Output: $F$
begin
    Remove each $u \in S$ where $w[u, v] < \tau$ for $v \in Q$
    $\mathbb{S}^* \leftarrow \emptyset$
    foreach $v \in S$ in descending order of $\alpha(v)$ do
        if $v$ is pruned by Accuracy Pruning then
            Continue
        $S_v \leftarrow \{ u \in S \mid d^E_V(u, v) \leq h \}$
        if $|S_v| < p$ then
            Continue
        if $\exists u \in S_v$ with $|L_u| < p$ then
            Add $v$ into $L_u$
        $\mathbb{S}_v \leftarrow \{u_1, .., u_p\}$, which are the $p$ vertices with
          maximum $\alpha(u_i)$ in $S_v$ (extracted with the aid
          from $L_v$)
        if $\Omega(\mathbb{S}_v) > \Omega(\mathbb{S}^*)$ then
            $\mathbb{S}^* \leftarrow \mathbb{S}_v$
    $F \leftarrow \mathbb{S}^*$
    return $F$
```

PROOF. Lemma 3 states that if vertex v is included in the optimal solution $\mathbb{S}^{OPT}$, then $\mathbb{S}^{OPT} \subseteq S_v$. Because HAE chooses the p vertices with maximum $\alpha(\cdot)$ in S_v as $\mathbb{S}_v$, there exists no other p-vertex subset of S_v with a larger objective value. Therefore, if $\mathbb{S}^{OPT} \subseteq S_v$, $\Omega(\mathbb{S}_v) \geq \Omega(\mathbb{S}^{OPT})$ must hold. On the other hand, $\Omega(\mathbb{S}^*) \geq \Omega(\mathbb{S}_v), \forall v \in S$, therefore, $\Omega(\mathbb{S}^*) \geq \Omega(\mathbb{S}^{OPT})$ holds. Moreover, Lemma 2 shows that Accuracy Pruning only prunes the examination of S_v if it cannot generate a better solution than $\mathbb{S}^*$. Please note that for any $v \in S$, $d^E_S(S_v) \leq 2h$. Therefore, F returned by algorithm HAE is no worse than the optimal solution with $d^E_S(F) \leq 2h$. The theorem follows. □

THEOREM 4. *HAE has time complexity $O(|R| + |S||E|)$.*

PROOF. HAE removes the SIoT objects that do not satisfy the accuracy constraint in $O(|R|)$ time. Sorting $v \in S$ in descending order of $\alpha(v)$ takes $O(|S|log|S|)$ time. That is, HAE spends $O(|R| + |S|log|S|)$ time for preprocessing.

HAE then considers each v in descending order of $\alpha(v)$. It first takes $O(|S| + |E|)$ time for Accuracy Pruning and extracting S_v for v. Then, HAE takes $O(|V|)$ time to check if there exists $u \in S_v$ with $|L_u| < p$ and $O(|V|)$ time to choose the p vertices u_i with the maximum $\alpha(u_i)$ from S_v. In summary, the time complexity of HAE is $O(|R| + |S|log|S|) + O(|S|(|S| + |E| + |V| + |V|)) = O(|R| + |S||E|)$. □

Although there is a bounded error h for F, in Section 6, we show that most F returned by HAE still satisfy the hop constraint with experiments conducted on real datasets.

5. ALGORITHM DESIGN FOR RG-TOSS

As proved in Section 3.2, RG-TOSS is NP-Hard and inapproximable within any ratio, indicating that RG-TOSS is very challenging due to the interplay of accuracy and communication robustness, i.e., the SIoT objects which have high accuracy may not have robust communications, and those with robust communication capability may not always have the optimized accuracy of the assigned tasks. To optimize the objective function, one simple approach is to greedily select F containing the p SIoT objects with the largest incident weights. However, this greedy approach may result in a set of SIoT objects that cannot communicate with each other at all. Another approach is to enumerate all the combinations of the SIoT objects. Although this brute-force approach can obtain the optimal solution, it incurs a prohibitively high computation complexity and thus is impractical.

To strike a good balance between solution quality and efficiency, in this section, we propose a polynomial-time algorithm to RG-TOSS, namely *Robustness-Aware SIoT Selection (RASS)* which can obtain good solutions very efficiently. RASS employs a bottom-up approach to construct different partial solutions while considering the accuracy and communication robustness simultaneously. To incrementally construct good partial solutions and lead to good solutions eventually, we propose an effective ordering strategy, called *Accuracy-oriented Robustness-aware Ordering (ARO)*, to prioritize the selections of SIoT objects into partial solutions. Moreover, we also propose effective pruning strategies, namely *Core-based Robustness Pruning (CRP) Accuracy-Optimization Pruning (AOP)*, and *Robustness-Guaranteed Pruning (RGP)*, which are based on our observations in different dimensions to avoid constructing partial solutions that can never grow into better solutions, in order to significantly reduce the computation time of RASS.

Specifically, to significantly reduce the computation time, RASS first employs a filter strategy to remove from G each SIoT object $u \in S$ not satisfying the accuracy constraint. Afterwards, RASS performs *Core-based Robustness Pruning (CRP)* to remove the SIoT objects in S which will not lead to feasible solutions. A $\widehat{k}$-core $C_{\widehat{k}}$ is a graph where each vertex $v \in C_{\widehat{k}}$ has degree at least $\widehat{k}$ [13]. A $\widehat{k}$-core $C_{\widehat{k}}$ is *maximal* if there does not exist another $\widehat{k}$-core that is a superset of $C_{\widehat{k}}$. Maximal $\widehat{k}$-core can be obtained in polynomial time.[3] In Core-based Robustness Pruning, RASS extracts the maximal k-core C_k from the graph formed by the SIoT objects and the corresponding social edge set, i.e., $G_S = (S, E)$, where k is the degree constraint. RASS then trims the SIoT objects from S which are not included in the maximal k-core C_k. The following lemma shows that the SIoT objects in $S \backslash C_k$ can be safely trimmed.

Lemma 4. ***Core-based Robustness Pruning.*** *Given maximal k-core $C_k \subseteq G_S$ and any feasible solution F to RG-TOSS, $(S \backslash C_k) \cap F = \varnothing$ must hold, indicating that SIoT objects in $S \backslash C_k$ can be safely trimmed.*

Proof. Suppose v is an SIoT object which is not included in the maximal k-core C_k, i.e., $v \in S \backslash C_k$. We prove this lemma by contradiction. Assume that F is a feasible solution and $v \in F$. As F is a feasible solution, $deg_F^E(u) \geq k, \forall u \in F$ must hold. Therefore, F is a k-core and $F \subseteq C_k$ holds (according to the definition of maximal k-core). Since $v \in F$, $v \in C_k$ must hold, which leads to a contradiction. Therefore, $v \notin F$ and v can be safely trimmed, $\forall v \in S \backslash C_k$. The lemma follows. □

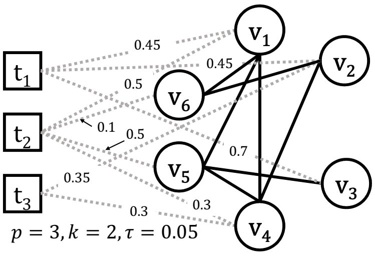


Figure 2: Running example of RG-TOSS

Consider the running example in Figure 2. Given the heterogeneous graph G with $p = 3$, $k = 2$, and $\tau = 0.05$. Since the maximal 2-core in $G_S = (S, E)$ is $\{v_1, v_2, v_4, v_5, v_6\}$, Core-based Robustness Pruning removes v_3 from S because v_3 will never be included in any feasible solution.

In algorithm RASS, each partial solution σ_i is defined as $\sigma_i = \{\mathbb{S}_i, \mathbb{C}_i\}$ where $\mathbb{S}_i$ is the solution set containing a set of SIoT objects, and $\mathbb{C}_i$ denotes the set of candidate SIoT objects that can be considered by the current partial solution σ_i. During the process of RASS, RASS maintains a priority queue $\mathbb{U}$ to store different partial solutions. Let $S = \{v_1, ..., v_{|S|}\}$. In the very beginning, RASS generates $|S|$ initial partial solutions and pushes them into $\mathbb{U}$, where each partial solution contains $\{\{v_i\}, \bigcup_{j \in [i+1, |S|]} v_j\}$ for each different $v_i \in S$. Return to the running example in Figure 2. In the beginning, priority queue $\mathbb{U}$ contains the following partial solutions: $\{\mathbb{S}_1 = \{v_1\}, \mathbb{C}_1 = \{v_2, v_4, v_5, v_6\}\}$,$\{\mathbb{S}_2 = \{v_2\}, \mathbb{C}_2 = \{v_4, v_5, v_6\}\}$,$\{\mathbb{S}_3 = \{v_4\}, \mathbb{C}_3 = \{v_5, v_6\}\}$. Please note that v_3 does not appear because it has been pruned by Core-based Robustness Pruning. Moreover, there is no partial solution $\{\mathbb{S}_4 = \{v_5\}, \mathbb{C}_4 = \{v_6\}\}$ because $|\mathbb{S}_4 \cup \mathbb{C}_4| < p = 3$. That is, even if we move all the candidate SIoT objects in $\mathbb{C}_4$ into $\mathbb{S}_4$, we still cannot form a feasible solution with exactly p SIoT objects. Similarly, $\{\mathbb{S}_5 = \{v_6\}, \mathbb{C}_5 = \phi\}$ does not exist in $\mathbb{U}$ as well.

At each step afterwards, RASS generates a new partial solution $\sigma' = \{\mathbb{S}', \mathbb{C}'\}$ as follows. RASS first pops from the priority queue $\mathbb{U}$ a partial solution $\sigma = \{\mathbb{S}, \mathbb{C}\}$ based on Accuracy-oriented Robustness-aware Ordering (ARO, will be detailed later), and RASS creates a copy of σ, i.e., σ'. Then for σ', RASS moves an SIoT object u with the maximum $\alpha(u)$ from its candidate SIoT object set $\mathbb{C}'$ into its solution set $\mathbb{S}'$. Therefore, σ' becomes a new partial solution, i.e., $\mathbb{S}' \backslash \mathbb{S} = \{u\}$.

Return to the running example in Figure 2, after initialization, assume ARO in RASS selects the partial solution $\sigma = \{\{v_1\}, \{v_2, v_4, v_5, v_6\}\}$ for expansion. RASS first creates a copy of σ, i.e., $\sigma' = \{\mathbb{S}' = \{v_1\}, \mathbb{C}' = \{v_2, v_4, v_5, v_6\}\}$. Since choosing v_2 for expanding $\mathbb{S}'$ does not satisfy ARO, RASS choose v_4 which satisfies ARO and has the maximum $\alpha(\cdot)$. Therefore, v_4 is moved to $\mathbb{S}'$ and $\sigma' = \{\{v_1, v_4\}, \{v_2, v_5, v_6\}\}$ is a new partial solution. RASS then removes v_4 from $\mathbb{C}$ of σ to avoid generating duplicate partial solution as σ' in the future, and pushes $\sigma = \{\{v_1\}, \{v_2, v_5, v_6\}\}$ back to the priority queue $\mathbb{U}$. Please note that, σ is always inserted back into the priority queue (unless $|\mathbb{S}| + |\mathbb{C}| < p$) because σ is able to generate other new partial solutions by expanding its solution set with other vertices. Moreover, in order to guarantee not generating duplicate partial solutions, the SIoT object that is moved from $\mathbb{C}'$ to $\mathbb{S}'$ is removed from $\mathbb{C}$ of σ. For example, v_4 is removed from $\mathbb{C}$ when σ is pushed back into $\mathbb{U}$.

[3] Please note that the maximal $\widehat{k}$-core may contain multiple connected components.

If the solution set $\mathbb{S}' \in \sigma'$ contains p SIoT objects, satisfies the degree constraint, and $\Omega(\mathbb{S}')$ is larger than the currently best solution $\mathbb{S}^*$, RASS updates $\mathbb{S}^*$ as $\mathbb{S}'$. If $\mathbb{S}'$ contains fewer than p SIoT objects, RASS inserts σ' into the priority queue $\mathbb{U}$. In the second round in our running example, RASS pops $\sigma = \{\{v_1, v_4\}, \{v_2, v_5, v_6\}\}$ according to ARO, and generates a new partial solution $\sigma' = \{\{v_1, v_4, v_5\}, \{v_2, v_6\}\}$. Since $\mathbb{S}' = \{v_1, v_4, v_5\}$ contains $p = 3$ SIoT objects and satisfies the constraints, and $\mathbb{S}'$ is the first feasible solution obtained so far, RASS sets $\mathbb{S}^* = \mathbb{S}'$. RASS pushes the original σ back to $\mathbb{U}$ (σ' does not have to be pushed back because $|\mathbb{S}'| = 3$). The number of expansions on partial solutions RASS performs is bounded by a parameter λ. After λ expansions of partial solutions, RASS outputs the best solution $\mathbb{S}^*$ as the solution. The setting of λ represents a trade-off between efficiency and solution quality. We will compare the performance of RASS under different λ values in the experimental results in Section 6.

In the following, we detail the important strategies employed in our framework. These strategies include *Accuracy-oriented Robustness-aware Ordering*, *Accuracy-Optimization Pruning* and *Robustness-Guaranteed Pruning*, which can significantly improve the efficiency and solution quality of the proposed RASS algorithm. The pseudo code of algorithm RASS is detailed in Algorithm 2.

```
Algorithm 2: Robustness-Aware SIoT Selection (RASS)
Input: $G = (T, S, E, R)$, $Q$, $k$, $p$, $\tau$, $\lambda$
Output: $F$
begin
    Remove each $u \in S$ where $w[u, v] < \tau$ for $v \in Q$
    $\mathbb{S}^* \leftarrow \emptyset$; $\mathbb{U} \leftarrow \emptyset$; $expand \leftarrow 0$
    CRP (Lemma 4) on $G_S = (S, E)$
    foreach $v_i \in S = \{v_1, .., v_{|S|}\}$ do
        push $\{\{v_i\}, \bigcup_{j \in [i+1, |S|]} v_j\}$ into $\mathbb{U}$
    while $expand < \lambda$ do
        $expand \leftarrow expand + 1$
        Pop $\sigma = \{\mathbb{S}, \mathbb{C}\}$ from $\mathbb{U}$ based on ARO
        if $\sigma$ can be pruned by AOP (Lemma 5) or RGP (Lemma 6) then
            Continue
        Copy $\sigma$ to $\sigma'$; Push $\sigma$ back into $\mathbb{U}$ if $|\mathbb{S}| + |\mathbb{C}| \geq p$
        $u \leftarrow \mathrm{argmax}_{x \in \mathbb{C}'} \alpha(x)$
        Move $u$ from $\mathbb{C}'$ to $\mathbb{S}'$
        if $|\mathbb{S}'| = p$ and $\Omega(\mathbb{S}') > \Omega(S^*)$ then
            $S^* \leftarrow S'$
        else if $|\mathbb{S}'| < p$ then
            push $\sigma'$ back to $\mathbb{U}$
    $F \leftarrow \mathbb{S}^*$
    return $F$
```

5.1 Accuracy-oriented Robustness-aware Ordering

The selection of partial solution σ from the priority queue to construct σ' is critical to the solution quality and algorithm efficiency. This is because a carefully selected σ can generate a good solution earlier, which can be used to prune other partial solutions afterwards. One simple approach, called *Accuracy Ordering*, is to select σ where its corresponding solution set $\mathbb{S}$ has the maximum $\Omega(\mathbb{S})$ (i.e., maximum accuracy), to expand into σ'.

Consider Figure 2, after initialization, Accuracy Ordering would select $\{\{v_1\}, \{v_2, v_4, v_5, v_6\}\}$ because $\{v_1\}$ has the maximum $\Omega(\mathbb{S})$. Moreover, this partial solution is copied and expanded into $\{\{v_1, v_2\}, \{v_4, v_5, v_6\}\}$ because v_2 has the maximum $\alpha(\cdot)$ in $\{v_2, v_4, v_5, v_6\}$. However, $\{v_1, v_2\}$ will not become a feasible solution because $p = 3$ and $k = 2$, i.e., requiring the SIoT objects in the solution to connect to each other. This example demonstrates that Accuracy Ordering does not consider the degree constraint and is inclined to obtain a set of SIoT objects without communication robustness, resulting in the generation of a large number of infeasible solutions.

To tackle the problem of Accuracy Ordering, we propose *Accuracy-oriented Robustness-aware Ordering (ARO)* to consider both accuracy and communication robustness simultaneously. The idea of ARO is to prioritize the selection of Accuracy Ordering with additional conditions of the communication robustness. Recall that Accuracy Ordering pops the partial solution $\sigma = \{\mathbb{S}, \mathbb{C}\}$ with the maximum $\Omega(\mathbb{S})$ from $\mathbb{U}$. Then σ' is constructed by moving the vertex $u \in \mathbb{C}$ which incurs the maximum $\alpha(u)$ to $\mathbb{S}$. In ARO, σ is selected from the priority queue only when $(\mathbb{S} \cup \{u\})$ (u incurs the maximum $\alpha(u)$ in $\mathbb{C}$) has sufficient communication robustness. In this case, ARO effectively guides RASS to expand good partial solutions which has high potential to satisfy the degree constraint.

Specifically, given a partial solution $\sigma = \{\mathbb{S}, \mathbb{C}\}$, let $\Delta(\mathbb{S}) = \frac{\sum_{v \in \mathbb{S}} deg_{\mathbb{S}}^{E}(v)}{|\mathbb{S}|}$ be the average inner degree of $\mathbb{S}$, and let u be the SIoT object in $\mathbb{C}$ which incurs the $\alpha(u)$. In ARO, we first assume that u is added to $\mathbb{S}$. Then, we examine if the communication robustness of the new set $\mathbb{S} \cup \{u\}$ is sufficiently large. After that, from those partial solutions that satisfy the communication robustness requirement, we select and pop the partial solution which incurs the maximum $\Omega(\cdot)$ for expansion. The communication robustness of $\mathbb{S} \cup \{u\}$ is considered sufficiently large if the following *Inner Degree Condition (IDC)* holds: $\Delta(\mathbb{S} \cup \{u\}) \geq |\mathbb{S} \cup \{u\}| - \frac{\mu \cdot |\mathbb{S} \cup \{u\}| + p - 1}{p - 1}$, where μ is a self-adjusting filtering parameter.

The filtering parameter μ is important for finding good feasible solutions quickly. Specifically, μ is set as $p - k - 1$ initially to provide a more strict filtering power when selecting SIoT objects into $\mathbb{S}$ to fulfill the degree constraint, i.e., when μ is larger, vertex u passes IDC when u has more inner degree in the current $\mathbb{S} \cup \{u\}$. When no SIoT object satisfies IDC with the current μ values, ARO decreases μ to lower the threshold until at least one vertex u satisfies IDC.

In our example shown in Figure 2, given $\sigma = \{\mathbb{S} = \{v_1\}, \mathbb{C} = \{v_2, v_4, v_5, v_6\}\}$, with $\mu = p - k - 1 = 0$, ARO avoids to select $v_2 \in \mathbb{C}$ to expand $\mathbb{S}$ (which would be chosen by the intuitive Accuracy Ordering) because $\Delta(v_1 \cup \{v_2\}) < 2 - 1$ does not satisfy the Inner Degree Condition. In fact, selecting v_2 by Accuracy Ordering would not expand σ into any feasible solution, but only costs unnecessary expansions. Instead, ARO considers the set of SIoT objects in $\mathbb{C}$ which satisfies Inner Degree Condition, i.e., $\{v_4, v_5, v_6\}$, and picks v_4 because v_4 incurs the maximum $\alpha(\cdot)$. As a result, ARO expands σ to $\sigma' = \{\{v_1, v_4\}, \{v_2, v_5, v_6\}\}$.

5.2 Pruning Strategies

The ordering strategy, i.e., ARO, described in Section 5.1 prioritizes the expansion of partial solutions that are likely to become good feasible solutions. It is expected that ARO is able to obtain the first feasible solution which follows the degree constraint much earlier than Accuracy Ordering because inner degrees are examined during the process. To reduce the examination of unnecessary partial solutions which will never become better feasible solutions, we further derive two pruning strategies, namely *Accuracy-Optimization Pruning (AOP)* and *Robustness-Guaranteed Pruning (RGP).*

Accuracy-Optimization Pruning (AOP) keeps tracks of the objective value of the currently best solution, i.e., $\Omega(\mathbb{S}^*)$ and removes the partial solutions that will never become a better solution than $\mathbb{S}^*$ by deriving the upper bound on the objective values of the partial solutions. Equipped with AOP, RASS is able to avoid unnecessary expansions of partial solutions and significantly reduces the computation time. On the other hand, Robustness-Guaranteed Pruning (RGP) considers the communication robustness of the partial solutions. RGP prunes the partial solutions (discards it from $\mathbb{U}$ directly) if they cannot grow into feasible solutions, i.e., satisfying the degree constraint. With RGP, algorithm RASS can avoid spending unnecessary computation resource on partial solutions that will not become feasible solutions.

Specifically, given the currently best solution $\mathbb{S}^*$ and its accuracy $\Omega(\mathbb{S}^*)$, Lemma 5 states Accuracy-Optimization Pruning.

LEMMA 5. ***Accuracy-Optimization Pruning (AOP).*** *Partial solution $\sigma = \{\mathbb{S}, \mathbb{C}\}$ can be removed from the priority queue $\mathbb{U}$ if $\sum_{v \in \mathbb{S}} \alpha(v) + (p - |\mathbb{S}|) \cdot \max_{u \in \mathbb{C}} \alpha(u) \leq \Omega(\mathbb{S}^*)$ holds.*

PROOF. The first term of the inequality is the total accuracy of the SIoT objects in $\mathbb{S}$, and the second term: $(p - |\mathbb{S}|) \cdot \max_{u \in \mathbb{C}} \alpha(u)$ is an upper bound on the total accuracy the current partial solution can achieve by adding $(p - |\mathbb{S}|)$ SIoT objects. Therefore, if the inequality holds, any solution constructed from the current partial solution σ will never have total accuracy larger than the currently best solution $\mathbb{S}^*$ and thus can be safely pruned. □

Return to the running example in Figure 2. After obtaining $\mathbb{S}^* = \{v_1, v_4, v_5\}$, assume that RASS is considering to expand $\sigma = \{\mathbb{S} = \{v_2\}, \mathbb{C} = \{v_4, v_5, v_6\}\}$. Since $\sum_{v \in \mathbb{S}} \alpha(v) = 0.8$ and $(p - |\mathbb{S}|) \cdot \max_{u \in \mathbb{C}} \alpha(u) = 2 \cdot 0.6$, $\sum_{v \in \mathbb{S}} \alpha(v) + (p - |\mathbb{S}|) \cdot \max_{u \in \mathbb{C}} \alpha(u) = 2.0 < \Omega(\mathbb{S}^*) = 2.05$. Therefore, σ can be directly removed from $\mathbb{U}$ without any further expansions.

On the other hand, Robustness-Guaranteed Pruning considers the degrees of the SIoT objects inside $\mathbb{S}$ and those outside $\mathbb{S}$ of a partial solution. The following Lemma 6 details RGP.

LEMMA 6. ***Robustness-Guaranteed Pruning (RGP).*** *Partial solution $\sigma = \{\mathbb{S}, \mathbb{C}\}$ can be removed from the priority queue $\mathbb{U}$ if one of the conditions holds: 1) $p - |\mathbb{S}| + \min_{v \in \mathbb{S}} deg^E_{\mathbb{S}}(v) < k$, or 2) $\sum_{v \in \mathbb{C}} deg^E_{\mathbb{C} \cup \mathbb{S}}(v) < k(p - |\mathbb{S}|)$.*

PROOF. For the first condition, $p - |\mathbb{S}|$ is the number of SIoT objects which will be added into $\mathbb{S}$, and $\min_{v \in \mathbb{S}} deg^E_{\mathbb{S}}(v)$ is the minimum inner degree of the SIoT objects in $\mathbb{S}$. Therefore, the left-hand-side of the first condition is the upper bound on the inner degree of the vertex with the minimum inner degree in $\mathbb{S}$. If the first condition holds, at least one SIoT object will not satisfy the degree constraint afterwards. The first condition considers the SIoT objects that have been moved into $\mathbb{S}$. For the second condition, it considers the SIoT objects that are in $\mathbb{C}$. For the Right-Hand-Side of the inequality, $(p - |\mathbb{S}|)$ is the number of SIoT objects that need to be moved from $\mathbb{C}$ into $\mathbb{S}$, and $k(p - |\mathbb{S}|)$ is the number of total vertex degrees the added vertices should have to become a feasible solution. Therefore, if the $\sum_{v \in \mathbb{C}} deg^E_{\mathbb{C} \cup \mathbb{S}}(v)$, i.e., the total vertex degrees $\mathbb{C}$ can provide, is fewer than $k(p - |\mathbb{S}|)$, the partial solution σ will never grow into a feasible solution. □

Return to Figure 2, assume RASS is now examining $\sigma = \{\{v_2\}, \{v_4, v_5, v_6\}\}$. Since $\sum_{v \in \mathbb{C}} deg^E_{\mathbb{C}}(v) = 1 + 1 + 0$, which is smaller than $k(p - |\mathbb{S}|) = 2 \cdot (3 - 1)$. Therefore, σ can be directly removed from $\mathbb{U}$ without further expansions. The following Theorem 5 summarizes the time complexity of RASS.

THEOREM 5. *RASS has time complexity $O(|R| + \lambda(|S| + \lambda)p^2)$.*

PROOF. RASS removes vertices which do not satisfy the accuracy constraint from S in $O(|R|)$ time. Core-based Robustness Pruning is performed in $O(|S|)$, and for initialization, RASS spends $O(|S|)$ to construct and push each partial solution $\{\{v_i\}, \bigcup_{j \in [i+1, |S|]} v_j\}$ into $\mathbb{U}$. Therefore, before RASS expands any partial solution, it takes $O(|R| + |S|)$ time.

To identify and pop a partial solution, since $\mathbb{U}$ has at most $(|S| + \lambda)$ partial solutions, RASS performs $O(|S| + \lambda)$ times Inner Degree Condition verification (which takes $O(p^2)$). That is, it costs $O((|S| + \lambda)p^2)$ to identify and pop the partial solution σ. It takes $O(p)$ and $O(|S|)$ time to examine Accuracy-Optimization Pruning and Robustness-Guaranteed Pruning, respectively, and $O(|S|)$ time to copy σ to σ'. Therefore, expanding a partial solution takes $O((|S| + \lambda)p^2)$ time. Since RASS expands at most λ partial solutions, expanding λ partial solutions takes $O(\lambda(|S| + \lambda)p^2)$. In summary, the time complexity of RASS is $O(|R| + \lambda(|S| + \lambda)p^2)$. □

6. EXPERIMENT

In this section, we first detail the preparation of the datasets used in our evaluation. Afterwards, we evaluate the performance of the proposed algorithms with real datasets. Since *BC-TOSS* and *BC-TOSS* are NP-hard, we first enumerate all the possible combinations on a small-scale dataset to derive the optimal solution, and compare it with our solutions. Moreover, to evaluate the efficiency and effectiveness of the proposed algorithms, a co-author network is transformed into an SIoT network, where each node in the network contains a skill set (detailed in Section 6.1). Finally, a user study with 100 people is conducted to compare manual coordination with the proposed *HAE* and *RASS*.

6.1 Experiment Setting

The first dataset, *RescueTeams*, contains a small set of the rescue and disaster response teams in Canada[4] and California, USA,[5] with 68 and 77 teams, respectively. We regard

[4] A part of the rescue and disaster response teams can be found on http://en.wikipedia.org/wiki/Canadian_Forces_Search_and_Rescue

[5] A part of the rescue and disaster response teams can be found on http://www.calema.ca.gov/Pages/default.aspx

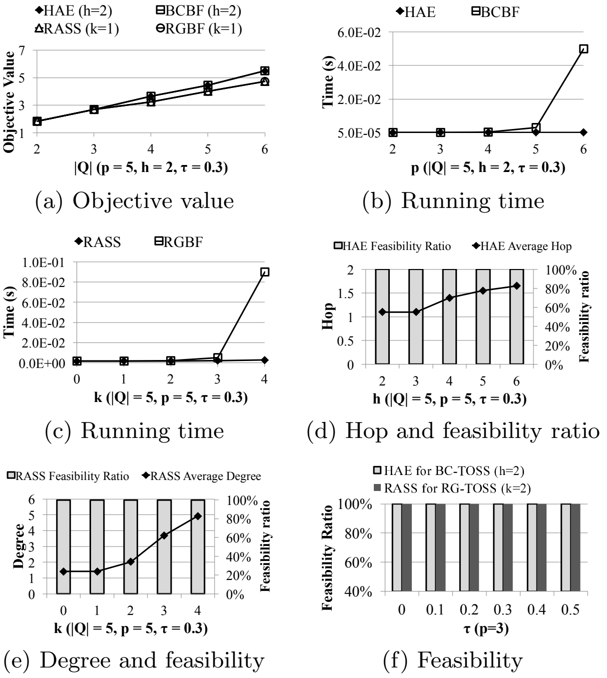
each team as a candidate SIoT object with possession of certain equipment representing proficiency in associated, e.g., a rescue and disaster response team with equipment A and B is viewed as a node in G with skills A and B. Moreover, the accuracies of edges are generated by uniform distribution ranged from 0 to 1. We also collect and analyze the spatial coordinates and characteristics of 34 and 32 disasters occurring in Canada and California, respectively, during the past 5 years to serve as the basis of queries and required skills in our evaluation. The types of disasters include wildfires, hurricanes, floods, earthquakes, and landslides. Due to the lack of social relations in the RescueTeam dataset, we create social links to the dataset based on the distance between two teams. We first sort all the pairwise distances in ascending order and select the top 50% to generate social edges.

Moreover, since there is no public large-scale SIoT dataset with specified tasks, to generate the input for the TOSS problem, we take Dataset *DBLP*, which contains $511,163$ nodes and $1,871,070$ edges, and only entries corresponding to the papers published in the areas of Database (DB), Artificial intelligence (AI), Data mining (DM), and Theory (T) conferences are kept. Only the authors who have at least three papers in the four areas are included, and each author is regarded as an SIoT object and the skills of authors are regarded as the tasks can be assigned to them. Moreover, we generate the skill set and social edges similar to [9]. Specifically, an author owns a skill (terms) if the term appears in at least two titles of papers that he has co-authored. We further generate the accuracy edges of author v_i by first counting the number of times each term appearing in titles of papers that he has co-authored and then normalizing it with the largest counts among all authors. Finally, two authors v_i and v_j are connected if they appear as co-authors in at least two papers in DBLP.

In the following, we compare *HAE* and *RASS* with two baselines. The first baseline is a brute-force method which enumerates all the feasible solutions for *BC-TOSS* (namely *BCBF*) and *RG-TOSS* (namely *RGBF*) to show the difference between the solutions derived from the proposed methods and optimal solutions. Moreover, we compare *HAE* and *RASS* with DpS [4]. DpS is an $O(|V|^{1/3})$-approximation algorithm for finding a p-vertex subgraph $H \subseteq S$ with the maximum density (the number of edges induced by H divided by $|H|$) on E without considering the query group, accuracy edges, hop or degree constraint. Finally, we implement the proposed algorithms, *HAE* and *RASS*, and invite 100 people from various communities, e.g., government, banks, hospitals, technology companies, schools, and businesses to join our user study, to compare the objective values and the time for answering *BC-TOSS* and *RG-TOSS* with manual coordination and proposed algorithms (i.e., *HAE* and *RASS*) to demonstrate the advantages of automatic query answering on SIoT. Each user is asked to plan 20 SIoT object selections for query answering with the query tasks. For the target graph, we sample a topology from Dataset *RescueTeams* and randomly connect edges to the query task with the weighting following the uniform distribution. All the experiments are implemented in an HP DL580 server with 4 Intel Xeon E7-4870 2.4 GHz CPUs and 1 TB RAM.

6.2 Performance Evaluation

6.2.1 *RescueTeams*



(a) Objective value

(b) Running time

(c) Running time

(d) Hop and feasibility ratio

(e) Degree and feasibility

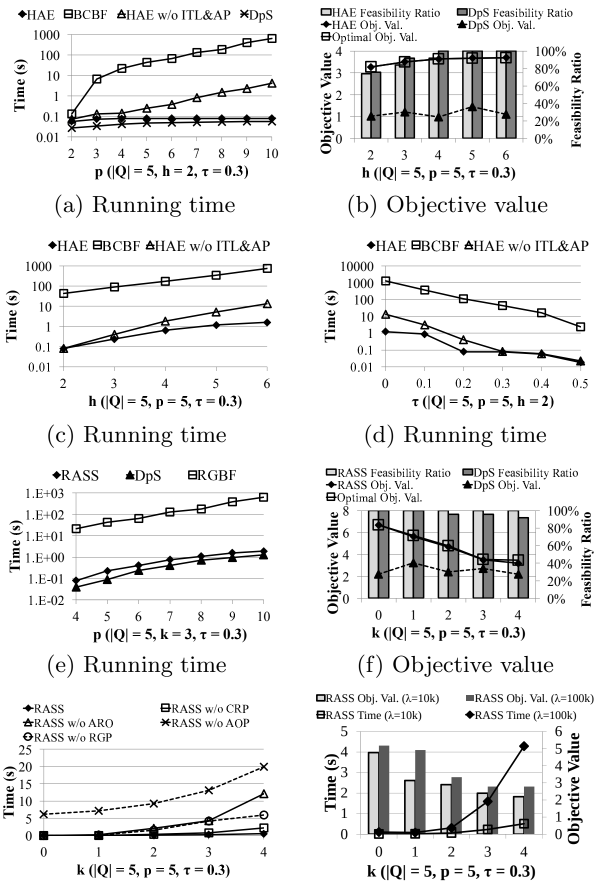
(f) Feasibility

Figure 3: Experiment results on *RescueTeams* dataset

In the following, we first compare the performance of *HAE* and *RASS* with baselines (*BCBF* and *RGBF*) on Dataset *RescueTeams*. *BCBF* and *RGBF* are brute-force algorithms which enumerate all the combinations of solutions, check the feasibility, and output the feasible solutions with the largest objective value. We randomly sample the query tasks 100 times and report the averaged results.

Figure 3(a) compares the objective values of *HAE* and *RASS* with *BCBF* and *RGBF*, respectively, for different query task sizes $|Q|$, where the budget constraint $p = 5$, hop constraint $h = 2$, and accuracy constraint $\tau = 0.3$. The results show that the objective value of the target group is proportional to the query group size $|Q|$. Moreover, *HAE* and *RASS* always derive the optimal objective value as $|Q|$ grows. The objective values of *HAE* are slightly larger than those of *RASS* since the constraint is looser and thus reduces the solution space. Figure 3(b) presents the running time for answering *BC-TOSS* with different budget constraints p. As p grows, the running time of *BCBF* significantly increases due to the large number of possible combinations, while the running time of *HAE* only slightly increases. On the other hand, Figure 3(c) presents the running time for answering *RG-TOSS* with different degree constraints k. *RASS* significantly outperforms *RGBF* as $|Q|$ grows.

In addition to objective values and running time, the feasibility ratio and average hop of *HAE* are reported in Figure 3(d). Although *HAE* slightly relaxes the hop constraint to derive the optimal solution with a bounded error, all the feasibility ratios w.r.t. different h are 100%. The average hop of solutions derived by *HAE* slightly increases as h grows, which implies that *HAE* finds optimal solutions of which SIoT objects are not far away from each other for providing data and transmission reliability. On the other hand, Figure 3(e) shows the feasibility ratio and average degree of *RASS*. All the feasibility ratios w.r.t. different K are 100%. Note



(a) Running time

(b) Objective value

(c) Running time

(d) Running time

(e) Running time

(f) Objective value

(g) Running time

(h) Running time

Figure 4: Experiment results on *DBLP* datasets

that the average degree of optimal solutions when $k = 0$ (no degree constraint) and $k = 1$ are close. This is because the rescue teams with different skills are usually not far from each other to cover all the rescue tasks within an area. As such, the average degree of the optimal solution without the degree constraint is more than 1, i.e., nodes are connected instead of being isolated. Figure 3(f) shows the feasibility ratio with different accuracy constraints from 0 to 0.5, and the results indicate the robustness of *HAE* and *RASS* since the feasibility ratios are all 100% given different accuracy constraints τ.

6.2.2 DBLP

We further evaluate and analyze the performance of the proposed algorithms on *DBLP* dataset. Figures 4(a)-(d) show the results for answering *BC-TOSS*. We report the results of *HAE* with three baselines: 1) brute-force method (*BCBF*), 2) Densest p-Subgraph (*DpS*) [4], and 3) *HAE* without Incident Weight Ordering with Top-p Object Lookup and Accuracy Pruning (*HAE w/o ITL&AP*).

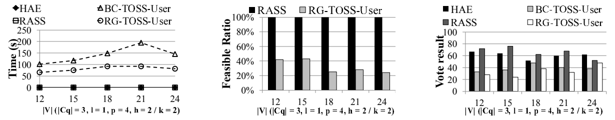
Figure 4(a) shows the running time with different budget constraint p, where $|Q| = 5$, $h = 2$, and $\tau = 0.3$. The result indicates that the running time of *HAE* is close to that of *DpS* but outperforms other baselines. The running time of *DpS* is the smallest since *DpS* only finds the densest p-subgraph without computing the feasibility of solutions. On the other hand, as p grows, the running time of *HAE* is much less than that of *HAE w/o ITL&AP*, which indicates the effectiveness of the lookup and pruning strategy. Figure 4(b) shows the objective values and feasibility ratios with different hop constraints given accuracy constraint $\tau = 0.3$. *DpS* slightly outperforms *HAE* in terms of feasibility ratio since *DpS* finds socially-tight groups which are inclined to satisfy the hop constraint. However, without considering the objective values, the objective values of *DpS* are much smaller than that of *HAE*, while the objective values of *HAE* are close to optimal. Figure 4(c) reports the running time with different hop constraints h. As h increases, the running time of all methods grows linearly, while the running time of *HAE* is still close to 1 second given $h = 6$. We further investigate the relationship between accuracy constraint τ and running time. The results manifest that the running time can be reduced when τ is large since the solution space is significantly reduced with large τ. However, if we set τ with a value close to 1, the solution space may become empty without any feasible solutions.

Figures 4(e)-(h) present the results for answering *RG-TOSS*. The results of *RASS* are compared with those of the brute-force method (*RGBF*) and *DpS*. Given $|Q| = 5$, $k = 3$, and $\tau = 0.3$, Figures 4(e) shows the running time with different budget constraints p. The results indicate that the proposed *RASS* outperforms *RGBF* by at least two orders. On the other hand, Figure 4(f) shows the objective values and feasibility ratios with different k. When the degree constraint k increases, the feasibility ratio of *RASS* is still 100% and outperforms *DpS* since ARO prioritizes the examination of partial solutions that will lead to feasible solutions. Note that *DpS* finds the densest subgraph but may not satisfy the degree constraint since most of the edges in the group may only be incident to some nodes, while the remaining nodes do not satisfy degree constraint. Meanwhile, the objective values of *RASS* are close to those of the optimal solutions.

We further conduct experiments on the running time and objective values with different k as shown in Figure 4(g). As the degree constraint becomes strict, i.e., the requirement of reliability in data transmission becomes high, the objective values become small since the cohesive requirement reduces the number of possible solutions and may not answer the task correctly. Moreover, as k increases, the running time of *RASS* also grows since the complexity of finding a cohesive group is high, e.g., cliques. Figure 4(h) shows the running time of *RASS*, *RASS* without Accuracy-oriented Robustness-aware Ordering (*RASS* w/o ARO), *RASS* without Core-based Robustness Pruning (*RASS* w/o CRP), *RASS* without Accuracy-Optimization Pruning (*RASS* w/o AOP), and *RASS* without Robustness-Guaranteed Pruning (*RASS* w/o RGP). The result manifests that Accuracy-Optimization Pruning (AOP) is the most effective because AOP precisely estimates the upper bounds of partial solutions and effectively prunes the partial solutions that cannot grow into better solutions.

6.2.3 User Study

Here we conduct a user study to show that human computation for *BC-TOSS* and *RG-TOSS* is time-consuming, while the objective values are not close to optimal even when the number of SIoT objects is small. Each user is assigned to solve *BC-TOSS* and *RG-TOSS* on 5 small SIoT networks with vertex set sizes 12, 15, 18, 21, and 24. To avoid confusing users with the complicated network structure, every vertex is labelled with an objective value, which is the sum-



(a) Running time (b) Feasible ratio (c) Vote result

Figure 5: User Study results

mation of the accuracy edge weights for the assigned tasks. For each instance, the query group size, i.e., $|Q|$, is 3, the budget constraint p is 4, the hop constraint is 2, and the degree constraint is 2.

Figure 5(a) compares manual coordination, *HAE*, and *RASS* in terms of running time. The result indicates that users spend from 50 to 200 seconds solving the *BC-TOSS* and *RG-TOSS* problem, while the running time for *HAE*, and *RASS* is close to 0. Moreover, the time of manual coordination for *BC-TOSS* is greater than that of *RG-TOSS* with different network sizes. However, as shown in Figure 5(b), the feasible ratio of manual coordination for*RG-TOSS* is small, especially for large network size, because it is difficult for users to check the degree constraint on network topology, while maximizing the summation of accuracy. The interplay between network topology and accuracy complicates *RG-TOSS*.

We also ask users to vote for the results of manual coordination, *HAE*, and *RASS*, as shown in Figure 5(c). The result manifests that users think our solutions are better as compared to the solutions found by themselves. Moreover, users think that *RASS* is more helpful since the feasibility examination on network topology for *RG-TOSS* problem is difficult. Therefore, it is desirable to deploy *HAE* and *RASS* as a service for automatic task-optimized group search, especially to address the need of a large group in a massive SIoT network nowadays.

7. CONCLUSIONS

In this paper, we propose and study a family of *Task-Optimized SIoT Selection (TOSS)* problems. To our best knowledge, this is the first paper that considers simultaneously the accuracy of performing tasks and the communication capability of SIoT objects. We study two different TOSS problems based on two different communication requirements, namely *BC-TOSS* and *RG-TOSS*. For BC-TOSS, we propose a polynomial-time algorithm with performance guarantee, and for RG-TOSS, we propose an efficient and effective algorithm that can obtain good solutions in polynomial time. We propose various ordering and pruning strategies for each algorithm to significantly reduce the computation time. Experimental results on real datasets show that our proposed algorithms outperform the other baselines.

8. ACKNOWLEDGEMENTS

This work was supported in part by Ministry of Science and Technology (MOST), Taiwan, under MOST 105-2218-E-007 -032 -MY2, and MOST 105-2218-E-002 -035 -.

9. REFERENCES

[1] B. I. Aydin, Y. S. Yilmaz, Y. Li, Q. Li, J. Gao, and M. Demirbas. Crowdsourcing for multiple-choice question answering. In *IAAI*, 2015.

[2] I.-R. Chen, F. Bao, and J. Guo. Trust-based service management for social internet of things systems. In *TDSC*, 2015.

[3] A. Fast, D. Jensen, and B. N. Levine. Creating social networks to improve peer-to-peer networking. In *KDD*, 2005.

[4] U. Feige, D. Peleg, and G. Kortsarz. The dense k-subgraph problem. In *Algorithmica*, 2001.

[5] A. V. Goldberg. Finding a maximum density subgraph. *Tech. Report No. UCB CSD 84/171*, 1984.

[6] N. Hamadeh, B. Daya, A. Hilal, and P. Chauvet. An analytical review on the most widely used meteorological models in forest fire prediction. In *TAEECE*, 2015.

[7] M. Kargar and A. An. Discovering top-k teams of experts with/without a leader in social networks. In *CIKM*, 2011.

[8] E. A. Kosmatos, N. D. Tselikas, and A. C. Boucouvalas. Integrating rfids and smart objects into a unified internet of things architecture. In *Advances in Internet of Things*, 2011.

[9] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *KDD*, 2009.

[10] W. Liu, W. Sun, C. Chen, Y. Huang, Y. Jing, and K. Chen. Circle of friend query in geo-social networks. In *DSAA*. Springer, 2012.

[11] R. J. Mokken. Cliques, clubs and clans. *Quality & Quantity*, 13(2):161–173, 1979.

[12] M. Nitti, R. Girau, and L. Atzori. Trustworthiness management in the social internet of things. *TKDE*, 26(5):1253–1266, 2014.

[13] S. B. Seidman. Network structure and minimum degree. In *Social Networks*, 1983.

[14] S. B. Seidman and B. L. Foster. A graph-theoretic generalization of the clique concept*. *Journal of Mathematical Sociology*, 6(1):139–154, 1978.

[15] C.-Y. Shen, D.-N. Yang, W.-C. Lee, and M.-S. Chen. Trustworthiness management in the social internet of things. *TKDD*, 10(47), 2016.

[16] H.-H. Shuai, D.-N. Yang, P. S. Yu, and M.-S. Chen. Willingness optimization for social group activity. In *VLDB*, 2014.

[17] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *KDD*, 2010.

[18] J. Surowiecki. *The wisdom of crowds*. Doubleday, 2004.

[19] S. Wasserman and K. Faust. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.

[20] D.-N. Yang, C.-Y. Shen, W.-C. Lee, and M.-S. Chen. On socio-spatial group query for location-based social networks. In *KDD*, 2012.

[21] L. Yao, Q. Z. Sheng, A. H. Ngu, H. Ashman, and X. Li. Exploring recommendations in internet of things. In *SIGIR*, 2014.

[22] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. Sybilguard: defending against sybil attacks via social networks. In *SIGCOMM*, 2006.

[23] Q. Zhu, H. Hu, J. Xu, and W.-C. Lee. Geo-social group queries with minimum acquaintance constraint. *arXiv preprint arXiv:1406.7367*, 2014.

Finding Socio-Textual Associations Among Locations

Paras Mehta
Freie Universität Berlin
Germany
paras.mehta@fu-berlin.de

Dimitris Sacharidis
Technische Universität Wien
Austria
dimitris@ec.tuwien.ac.at

Dimitrios Skoutas
IMIS, Athena R.C.
Greece
dskoutas@imis.athena-innovation.gr

Agnès Voisard
Freie Universität Berlin
Germany
agnes.voisard@fu-berlin.de

ABSTRACT

An increasing amount of user-generated content on the Web is geotagged. This often results in the formation of user trails, e.g., sequences of photos, check-ins, or text messages, that users generate while visiting various locations. In this paper, we introduce and study the problem of identifying sets of locations that are strongly associated under social and textual criteria. We say that a location set is associated with a set of keywords if there exists a user with posts around these locations whose textual descriptions cover all keywords. We measure the strength of this association by the number of users with posts that support it. Although the problem reminisces frequent itemset mining, we show that our support measure does not satisfy the necessary anti-monotonicity property, which is used to effectively prune the search space. Nonetheless, by studying the characteristics of the support measure, we are able to devise an efficient approach. We present a basic and two optimized algorithms, exploiting an inverted or a spatio-textual index to increase efficiency. Finally, we conduct an experimental evaluation using geotagged Flickr photos in three major cities. From a qualitative perspective, the results indicate that the introduced type of query returns meaningful and interesting location sets, which are not discovered by other existing approaches. Furthermore, the proposed optimizations and the use of appropriate indexes significantly reduce computation time.

1. INTRODUCTION

With the increasingly widespread use of mobile GPS-enabled devices and social networks, the amount of geotagged content on the Web is constantly growing. A user moving around a city may upload photos, post tweets, or check in at various locations, generating a *digital trail* of activities, which can be represented as a sequence of geotagged posts. Such publicly available trails enable the analysis and extraction of location associations that are implicitly defined by the activities of city dwellers or visitors. In turn, these associations can be used to build smarter location-based services and better understand how people experience their urban environment.

In this work, we seek to find *Socio-Textual Associations* (STAs) among locations that are strongly supported by a corpus of geotagged social media content. Given a set of keywords, we say that a group of locations are socio-textually associated if a user has posts near each of these locations, and the combined keyword set of these posts contains all query keywords. The more people make an association, i.e., the stronger its support in the corpus is, the likelier it is that there exists a latent thematic connection among the locations.

Compared to previous works that search for connections among a group of locations, our work has the distinguishing and novel aspect that it considers *social and textual criteria in unison* to define associations. In one line of work (e.g., [12, 10, 15, 3, 19, 23]), which we term *Location Patterns* (LP), the objective is to determine groups, patterns or sequences of locations (or regions) that are frequent in terms of purely social criteria, i.e., how many people support them. Since the process ignores the textual aspect, the identified locations are not semantically characterized or distinguished, and thus there is no mechanism to explore or exploit the resulting groups under a thematic context. For instance, this limits queries to finding the overall most frequent sequence of locations in a given area, or the most frequent Point of Interest (POI) to visit next. Even though one could easily enrich locations with textual information *after* the mining process, say to support recommending the most frequent restaurant to visit next, the locations remain only socially associated, and not thematically, because the computed frequencies still ignore the textual aspect.

Another related line of work is the *Collective Spatial Keyword* (CSK) query [21, 4], where locations are grouped according to textual criteria (i.e., they must collectively cover the given keywords) and spatial criteria (i.e., they must be close to each other and/or to the user's location). Thus, the optimization objective is an aggregate spatial distance, instead of some evidence-based frequency metric. In other words, the strength of the association among a valid group of locations (i.e., one that covers all keywords) is defined by spatial proximity alone. Again, this *proximity-based* approach fails to establish a thematic connection evidenced by users' behavior. For example, the fact that there is a restaurant next to an art exhibition venue, does not necessarily imply that art-loving people would find this particular restaurant attractive, unless such a connection is indeed supported by a large number of posts, from the same users, containing, for example, both keywords "*art*" and "*restaurant*" around these locations. As a matter of fact, if a strong thematic association among nearby locations exists, our problem formulation will certainly capture it.

A rather straightforward way to associate locations with keywords

10.5441/002/edbt.2017.12

Table 1: Categorization of Existing Work and Ours

Line of Work	Information Exploited			Optimization
	Spatial	Textual	Social	Objective
Location Patterns (LP) [3, 10, 12, 15, 19, 23]	×		×	frequency
Collective Spatial Keyword (CSK) [4, 21]	×	×		proximity
Aggregate Popularity (AP)	×	×	×	popularity
Socio-Textual Associations (STA)	×	×	×	frequency

according to users' behavior is based on rank aggregation [8]. For each keyword, consider a ranking of locations according to the keyword popularity, i.e., the number of posts that contain it. Then, to derive a group of locations that is most associated with a set of keywords, one can simply collect the most popular location for each keyword. This approach, which we call *Aggregate Popularity* (AP), has the advantage that individual locations are strongly associated with their respective keywords, but the location set as a whole may lack a strong socio-textual association. Indeed, each location may be popular for a different type of users, hence there may be no significantly sized population for which all these locations are popular. Exactly as in the case of proximity-based associations, if a strong thematic association among popular locations exists, our socio-textual approach will discover it.

Another differentiating trait of our work is that we consider the textual information that is included in the posts themselves, and do not rely on an external categorization of locations or POIs. The reason is that we seek to exploit the *wisdom of the crowd* to also determine textual relevance, in addition to quantify the strength of derived associations. Nonetheless, our methods can be readily adapted to take into account external textual descriptions as well.

To better frame our contribution with respect to previous works, Table 1 summarizes all approaches according to the type of information they exploit, i.e., spatial, textual, or social (user id), as well as the objective they optimize for. Mining location patterns does not exploit textual information, and seeks for groups of locations that maximize the frequency with which they co-appear among users' trails. On the other hand, collective spatial keyword queries ignore the social aspect, and look for location sets that maximize their proximity (to each other and/or a target location) subject to the constraint that they cover given keywords. An approach based on aggregating popularity considers all types of information available, and strives to include locations that are individually popular for some keyword and collectively cover given keywords. Our work also considers all types of information, but optimizes for a frequency metric that counts co-appearances of locations under a certain theme/topic/context, which is defined by the given keywords.

As an example, consider a search for locations in Berlin using the keywords "*wall*", "*art*" and "*restaurant*". Figure 1 depicts the results returned by different alternative approaches for combining locations to satisfy these keywords. Our socio-textual based approach returns the following location set as the top result (star-shaped markers): ⟨ "East Side Gallery", "Hackescher Markt" ⟩. The former is a portion of the Berlin wall covered with paintings, hence hosting many posts with the keywords "*wall*" and "*art*". The latter is a popular square in the city center, hosting also a series of restaurants frequently visited by tourists and travelers. As it turns out, these locations are neither the most popular ones for each individual keyword (see locations with circle-shaped markers, returned by the AP approach) nor close to each other. Yet, they reveal an interesting association, hinting to the fact that many travelers that have visited or plan to visit the Wall, being interested in art, tend to also prefer restaurants located at Hackescher Markt.

Furthermore, a search based on CSK identified around 350 singleton locations, for which there exists at least one user with posts containing all query keywords. One of these results is illustrated in Figure 1 (square-shaped marker). It is not straightforward how to select the best among these results; in fact, several of them may even be due to outliers or noise, which are inherent to crowdsourced content. Since a CSK query does not take frequency into account, it is better suited for cases where the query terms refer to (curated) POI categories, while being error prone and sensitive to outliers when searching on raw tags. On the other hand, the top result based on AP consists of Brandenburg Gate (for "*wall*"), a famous monument close to where the Berlin wall used to pass; the intersection of Gneisenaustr. and Mehringdamm streets (for "*restaurant*"), a place with many popular restaurants; and Stattbad Wedding (for "*art*"), a former well-known art venue. Each of these locations is popular for the respective query keyword, but they do not represent any strong shared interest between the people visiting them.

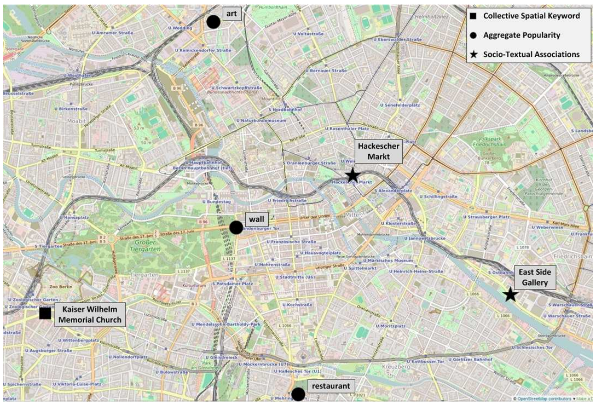

■ Collective Spatial Keyword
● Aggregate Popularity
★ Socio-Textual Associations
art
wall
restaurant
Hackescher Markt
East Side Gallery
Kaiser Wilhelm Memorial Church

Figure 1: Example of location sets retrieved for keywords "*wall*", "*art*" and "*restaurant*" in Berlin.

Existing algorithms for related problems cannot be used to extract socio-textual associations. Although our problem seems similar to mining frequent location patterns, the requirement for the locations to collectively cover certain keywords significantly complicates the problem, as we discuss in Section 4. Specifically, our notion of support (frequency) for a location set *does not exhibit the anti-monotonicity property* necessary to apply an Apriori-like algorithm [1]. Briefly, such a property would allow for early pruning of location sets that cannot be extended to produce valid results. Practically, the implication is that a naïve algorithm for even a relatively small-sized city-level dataset, with around 20,000 distinct locations, would need to investigate more than 10^{13} sets of three locations.

Nevertheless, by studying the problem characteristics, we are able to introduce a weaker notion of support that (1) exhibits anti-monotonicity, and (2) is an upper bound on the actual support of location sets. Armed with these two properties, we then introduce a methodology to efficiently identify location sets with strong socio-textual associations. Moreover, we study three different implementations of this methodology, each having its own merits. In the simplest, we assume that no pre-processing is allowed and that no index structure is available. We then present a method based on a simple off-the-shelf inverted index, and demonstrate how it can significantly speed up processing. The only caveat is that the association of locations with nearby posts is assumed to be known beforehand. Finally, leveraging the recent advances in spatio-textual indices, we devise an algorithm that exploits their general functionality. In particular, we consider the state-of-the-art I^3 index [22], which we also extend further to derive an even faster approach. Compared to the inverted index approach, the spatio-textual index methods allow to define the association of locations with nearby posts dynamically, which causes an overhead in execution time but provides higher flexibility.

In addition, we consider the problem of ranking socio-textually associated location sets instead of relying on a user-specified minimum support threshold. Thus, we directly address the problem of identifying the k most strongly associated location sets. We describe a general methodology, and propose algorithms that build upon their threshold-based counterparts.

The main contributions of our work are summarized below:

- We introduce and formally define the problem of finding socio-textually associated location sets.
- We study the problem characteristics and introduce a general framework based on a weaker support measure, which satisfies the desirable anti-monotonicity property.
- We present a basic algorithm, and two efficient algorithms that exploit an inverted index and a spatio-textual index, respectively, to significantly speed up computation.
- We consider the ranking variant of the problem, and discuss the necessary adaptations to all proposed algorithms.
- We present results from an experimental evaluation using real-world data from geolocated Flickr photo trails in three major cities.

The rest of the paper is structured as follows. In the next section, we present related work. Then, we formally define the problems in Section 3, and study their characteristics in Section 4. Following this analysis, we present our algorithms in Section 5, and extend them to the top-k variant in Section 6. Finally, Section 7 presents our experimental evaluation, and Section 8 concludes the paper.

2. RELATED WORK

Next, we review related work on the topics of mining frequent locations from geotagged posts and spatial keyword search.

2.1 Mining Geotagged Posts

Several approaches analyze trails of geotagged posts, mainly photos, to extract interesting Location Patterns (LP), such as scenic routes or frequently traveled paths. A typical methodology is to use a clustering algorithm to extract landmark locations from the original posts, and then apply sequence pattern mining.

In [12], clustering is first used to identify POIs; then, association rule mining is applied to extract associative patterns among them. In [10], each photo is first assigned to a nearby POI, whereas, for the remaining ones, a density-based clustering algorithm is applied to generate additional locations. Then, a travel sequence is constructed for each user, and sequence patterns are mined from these individual travel sequences. In [15], kernel vector quantization is used to find clusters of photos; then, routes are defined as sequences of photos from the same user, and patterns are revealed by applying hierarchical clustering on routes using the Levenshtein distance. In [3], a trajectory pattern mining algorithm is applied on geotagged Flickr photos to identify frequent travel patterns and regions of interest. In [16], a clustering method is applied on geotagged photos to identify and rank popular travel landmarks.

Geotagged photos have been used to measure the attractiveness of road segments in route recommendation. A tree-based hierarchical graph is used in [24] to infer users' travel experiences and interest of a location from individual sequences. Considering the transition probability between locations, frequent travel sequences are identified. Ranking trajectory patterns mined from sequences of geotagged photos is investigated in [19]. The mean-shift algorithm extracts locations from the original GPS coordinates of the photos; then, the PrefixSpan algorithm identifies the frequent sequential patterns, which are ranked based on user and location importance. In [23], density-based clustering is used to identify regions of attractions from trails of geotagged photos; then, the Markov chain model is applied to mine transition patterns among them.

Other efforts have focused on automatic trip planning or personalized scenic route recommendations based on geotagged photo trails, taking into account user preferences, current or previous locations, and/or time budget (e.g., [13, 17]). In [6], individual photo streams are integrated into a POI graph, and itineraries are constructed based on POI popularity, available time, and destination. In [14], users' traveling preferences are learned from their travel histories in one city, and then used to recommend travel destinations and routes in a different city. In [11], a set of location sequences that match the user's preferences, present location, and time budget, are computed from individual itineraries. From a different perspective, a Bayesian approach is applied in [2] to test different hypotheses about how photo trails are produced. Various assumptions are assessed, e.g., that users tend to take photos close to the city center, near POIs, close to their previous location, or a mixture of these.

Similar to the works presented above, we also select locations that appear frequently in users' posts. However, in our case these locations should be strongly associated with a given set of keywords, a requirement which complicates the search.

2.2 Spatial Keyword Search

Spatial keyword search involves queries that comprise a user location and a set of keywords. Both the spatial and the textual parts can be applied as boolean filters or as ranking criteria. For example, the query may retrieve all relevant objects within a specified distance from the given location, or rank them based on their proximity to it; similarly, it may retrieve all objects containing one or more of the query keywords, or rank them based on relevance. A comprehensive survey of existing approaches is presented in [5].

These efforts focus on combining spatial and textual indices into hybrid ones. Accordingly, they can be characterized as text-first or space-first [7]. For example, the IF-R*-tree uses an inverted file where the postings in each inverted list are indexed by an R-tree; on the other hand, the R*-tree-IF employs an R*-tree where inverted files are attached to each leaf node [25]. More recent methods have focused on retrieving top-k objects, ranked by an aggregate score combining both spatial proximity and textual relevance [22, 20].

More closely related to our work are Collective Spatial Keyword (CSK) queries, such as the mCK query [21]. Given m keywords, it retrieves a set of spatio-textual objects that are as close to each other as possible and collectively contain all keywords. A similar variant is defined in [4], where the retrieved objects need to be as close to the user location as possible, and, optionally, in close proximity to each other.

In our work, we search for a set of locations that cover all given keywords. However, instead of optimizing for spatial proximity, we seek to maximize their co-occurrence in user trails. Thus, the approach for addressing the problem is fundamentally different.

3. PROBLEM DEFINITION

Assume a database of posts $\mathcal{P}$ made by users $\mathcal{U}$. Each post $p \in \mathcal{P}$ is a tuple $p = \langle u, \ell, \Psi \rangle$, where $p.u \in \mathcal{U}$ is the user that made the post, $p.\ell = (lon, lat)$ is the geotag (location) of the post, and $p.\Psi$ is a set of keywords that characterize it. We use $\mathcal{P}_u$ to denote all posts of user u, i.e., $\mathcal{P}_u = \{p \in \mathcal{P} : p.u = u\}$. Furthermore, assume a database of locations $\mathcal{L}$. These may correspond to the posts' locations, or, for generality, may also be defined independently of $\mathcal{P}$. For instance, one may use a POI database to populate $\mathcal{L}$, or apply a clustering algorithm on the posts' geotags and then construct $\mathcal{L}$ from the cluster centroids. Thus, we reserve the term *location* for

Table 2: Notation

Symbol	Definition
$p, \mathcal{P}$	post, database of posts
$u, \mathcal{P}_u$	user, posts of user
$\ell, L, \mathcal{L}$	location, set of locations, database of locations
ψ, Ψ	keyword, set of keywords
$\mathcal{U}_{L\Psi}$	set of users supporting (L, Ψ)
$\mathcal{U}_{L\widetilde{\Psi}}$	set of users weakly supporting (L, Ψ)
$\mathcal{U}_{\Psi}$	set of users relevant to Ψ
$sup(L, \Psi)$	support of (L, Ψ)
$w\_sup(L, \Psi)$	weak support of (L, Ψ)
$rw\_sup(L, \Psi)$	relevant and weak support of (L, Ψ)
σ	support threshold

a member of $\mathcal{L}$, and refer to a post's location as its *geotag*. Table 2 summarizes the most important notation.

Locations are the principle objects in our work. We seek to identify sets of locations that are *strongly associated* with a set of keywords. To define this association, we first introduce the concepts of locality and (textual) relevance for a post.

Definition 1 (Local Post). *A post p is* local *to location ℓ if the post's geotag is within distance ϵ to ℓ, i.e., if $d(p.\ell, \ell) \leq \epsilon$, where d is a distance metric (e.g., Euclidean).*

Definition 2 (Relevant Post). *A post p is* relevant *to keyword ψ if the post's keyword set contains ψ, i.e., if $\psi \in p.\Psi$.*

Posts associate locations with keywords. These associations are bestowed by users themselves, as opposed, for example, to a specific POI categorization made by a particular source; thus, they capture the wisdom of the crowd. To model the relationships between users' posts, locations, and keywords, we introduce a bipartite graph, where the two types of vertices correspond to keywords and locations, while edges correspond to users' posts.

Definition 3 (Association Graph). *The* Association Graph *is a bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \Psi \cup \mathcal{L}$ and $\mathcal{E} \subseteq \Psi \times \mathcal{L}$, such that an edge $e = (\psi, \ell)$ exists iff there exists at least one post p which is local to ℓ and relevant to ψ; moreover, e is labeled with the set of users that have made such posts.*

Figure 2 shows a running example with the posts of five users $u_1, \ldots, u_5$ around three locations ℓ_1, ℓ_2, ℓ_3, containing two keywords ψ_1, ψ_2. Post p_{ij} denotes the j-th post of the i-th user. For instance, post $p_{12} = \langle u_1, \ell_2, \{\psi_1, \psi_2\}\rangle$ of user u_1 is local to location ℓ_2 and relevant to keywords ψ_1 and ψ_2. The resulting Association Graph is depicted in Figure 3.

The association between a keyword and a location is explicit, and its strength can be quantified by the number of users making it. For example, three users have associated keyword ψ_1 with location ℓ_3 in the running example. On the other hand, the association between sets of keywords and sets of locations is not immediately apparent, e.g., what the textual description of the location set $\{\ell_1, \ell_2\}$ should be. If it is simply the set of keywords that have an edge towards the location set, then how do we quantify its strength if different users have made different associations? The location set should be strongly associated with a set of keywords not because there exist edges with multiple users in the Association Graph, but because there exists *a large number of users that agree on this association*. Therefore, the key question to answer is when a user supports an association between a location set and a keyword set.

Definition 4 (Supporting User). *A user u* supports *the association between a location set L and keyword set Ψ, denoted as $u \in \mathcal{U}_{L\Psi}$, if:*

- *for each keyword $\psi \in \Psi$, the user has made a post relevant to ψ and local to a location $\ell' \in L$, i.e., every $\psi \in \Psi$ is connected via a u-labeled edge to some $\ell' \in L$; and*
- *for each location $\ell \in L$, the user has made a post local to ℓ and relevant to a keyword $\psi' \in \Psi$, i.e., every $\ell \in L$ is connected via a u-labeled edge to some $\psi' \in \Psi$.*

	Locations		
Users	ℓ_1	ℓ_2	ℓ_3
u_1	$p_{11} : \{\psi_1\}$	$p_{12} : \{\psi_1, \psi_2\}$	$p_{13} : \{\psi_1\}$
u_2	$p_{21} : \{\psi_1\}$	$p_{22} : \{\psi_1\}$	
u_3	$p_{31} : \{\psi_2\}$	$p_{32} : \{\psi_1\}$	$p_{33} : \{\psi_1\}$
u_4		$p_{42} : \{\psi_2\}$	$p_{43} : \{\psi_1\}$
u_5	$p_{51} : \{\psi_1, \psi_2\}$		

$$L = \{\ell_1, \ell_2\}, \quad \Psi = \{\psi_1, \psi_2\}$$
$$\mathcal{U}_{L\Psi} = \{u_1, u_3\}, \quad \mathcal{U}_{L\widetilde{\Psi}} = \{u_1, u_2, u_3\}$$
$$\mathcal{U}_{\Psi} = \{u_1, u_3, u_4, u_5\}, \quad \mathcal{U}_{\widetilde{L}\Psi} = \{u_1, u_3, u_5\}$$

$$sup(L, \Psi) = 2, \;\; w\_sup(L, \Psi) = 3, \;\; rw\_sup(L, \Psi) = 2$$

Figure 2: Running example.

Hence, a user supports association (L, Ψ) if her posts connect each keyword in Ψ to some location in L, and, vice versa, each location in L to some keyword in Ψ. This implies a tight coupling between *all* keywords and *all* locations, according to the user.

An association extracted from a user's posts between a keyword set and a location set could be arbitrary. After all, the content of a post is not always related to the location where it was made, and crowdsourced content is known to be characterized by errors and noise. Hence, an association acquires credence by the number of users supporting it. Accordingly, we use this to measure the strength of a keywords-locations association.

Definition 5 (Support). *The* support *of an association between a location set L and keyword set Ψ is the number of users supporting (L, Ψ), i.e., $sup(L, \Psi) = |\mathcal{U}_{L\Psi}|$.*

Returning to our example, user u_1 supports the location set $L = \{\ell_1, \ell_2\}$ and keyword set $\Psi = \{\psi_1, \psi_2\}$. For instance, post p_{11} (resp. p_{12}) is relevant to ψ_1 (resp. ψ_2) and local to some location among L; hence the first condition is satisfied; similarly, the second condition is also satisfied. It is not hard to see that the conditions are also satisfied for user u_3. Therefore, $sup(L, \Psi) = 2$.

We can now formally state the objective of this work. Given a set of keywords, we formulate two variants, one that retrieves all associations above a support threshold, and one that retrieves the k most strongly supported associations.

Problem 1 (Frequent Socio-Textual Associations). *Given a keyword set Ψ and a support threshold σ, identify all the location sets, up to cardinality m, that have support above σ.*

Problem 2 (Top-k Socio-Textual Associations). *Given a keyword set Ψ, identify k location sets, up to cardinality m, that have the highest support.*

The restriction on the cardinality of the location set is because, as explained in Section 4, adding more locations can increase the support of the set.

4. OBSERVATIONS AND APPROACH

Our approach is based on some key observations regarding the intrinsic characteristics of the studied problems. In fact, the stated problems reminisce the frequent itemset problem; however, the key

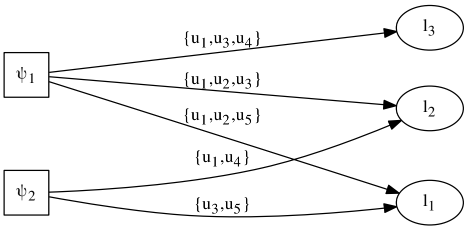


Figure 3: Association Graph for the running example.

difference here is that the introduced support function does not have the necessary anti-monotonicity property which allows for applying the Apriori principle. Given two sets X, Y, this property states that if $X \subseteq Y$, then $sup(X) \geq sup(Y)$. In other words, adding more items to a set cannot increase its support. However, the support introduced in Definition 5 does not exhibit this property.

Theorem 1. *The support of a location set L and a keyword set Ψ is not anti-monotonic with respect to the location set, i.e., there exist two location sets $L \subseteq L'$ and a keyword set Ψ, such that $sup(L, \Psi) < sup(L', \Psi)$.*

Proof. We prove via an example. Assume three keywords, four locations, and two users who have made posts in exactly those locations, as shown below:

	ℓ_1	ℓ_2	ℓ_3	ℓ_4
u_1	ψ_1	ψ_2	ψ_3	ψ_1
u_2	ψ_3	ψ_1	ψ_1	ψ_2

Consider the keyword set $\Psi = \{\psi_1, \psi_2, \psi_3\}$. Notice that only user u_1 supports location set $L = \{\ell_1, \ell_2, \ell_3\}$, i.e., $sup(L, \Psi) = 1$. On the other hand, both users support location set $L' = \{\ell_1, \ell_2, \ell_3, \ell_4\}$, i.e., $sup(L', \Psi) = 2$. In fact, any 3-location set in this example has support at most 1. □

As a matter of fact, the support of a location set and a keyword set can increase or decrease with respect to the location set. Despite this negative result, we devise an efficient filter-and-refine approach, where the filtering step exploits a weaker support measure.

Definition 6 (Weakly Supporting User). *A user u* weakly supports *a given location set L and keyword set Ψ, denoted as $u \in \mathcal{U}_{L\widetilde{\Psi}}$, if for each location $\ell \in L$, the user has made a post local to ℓ and relevant to a keyword in Ψ.*

The difference with respect to Definition 4 is that only the second condition applies. In other words, in the Association Graph, there must exist edges associating each one of the locations in L with keywords from Ψ, but without necessarily involving all keywords in Ψ. Accordingly, we define the notion of weak support.

Definition 7 (Weak Support). *The* weak support *of a given location set L and keyword set Ψ is the number of users weakly supporting (L, Ψ), i.e., $w\_sup(L, \Psi) = |\mathcal{U}_{L\widetilde{\Psi}}|$.*

In our example, user u_2 weakly supports (L, Ψ), where $L = \{\ell_1, \ell_2\}$ and $\Psi = \{\psi_1, \psi_2\}$. For both locations, u_2 has local posts (p_{21} and p_{22}) that are relevant to at least one keyword (ψ_1). In addition, users u_1, u_3 also weakly support the same location set and keyword set. On the other hand, u_4 and u_5 do not, as they do not have posts local to both locations. Therefore, $w\_sup(L, \Psi) = 3$.

Our filter and refine approach hinges on two properties of the weak support. The first is its anti-monotonicity, while the second is that it provides an upper bound for the support of an association.

Lemma 1. *The weak support of a location set and a keyword set is anti-monotonic with respect to the location set, i.e., for any two location sets $L' \subseteq L$ and keyword set Ψ, it holds that $w\_sup(L', \Psi) \geq w\_sup(L, \Psi)$.*

Proof. We show that any user u that does not weakly support (L', Ψ) cannot weakly support (L, Ψ). Assume otherwise, meaning that for each location in L there exists a post of u that is local to that location and relevant to the set Ψ. Trivially, this property also holds for any location in $L' \subseteq L$. Therefore, u must also support (L', Ψ) — a contradiction. □

Lemma 2. *The support of location set L and keyword set Ψ is not greater than their weak support, i.e., $sup(L, \Psi) \leq w\_sup(L, \Psi)$.*

Proof. We show that any user u that supports (L, Ψ) also weakly supports (L, Ψ). As per Definition 4, u has made a post local to each location in L and relevant to a keyword in Ψ (second condition). Therefore, the condition of Definition 8 applies, and u must also weakly support (L, Ψ). □

Returning to the example, users u_1, u_2, u_3, u_5 weakly support (L', Ψ), where $L' = \{\ell_1\}$. Hence, as per Lemma 1, $w\_sup(L', \Psi) \geq w\_sup(L, \Psi)$. Moreover, as per Lemma 2, we have seen that the weak support of (L, Ψ) is one more than its support. Based on these lemmas, we can derive the following important property.

Theorem 2. *If the weak support of a location set L and a keyword set Ψ is less than σ, then the support of any location set $L' \supseteq L$ and Ψ cannot be more than σ.*

Proof. The premise suggests that $\sigma > w\_sup(L, \Psi)$. From Lemma 1 we have that $w\_sup(L, \Psi) \geq w\_sup(L', \Psi)$, while from Lemma 2 we get $w\_sup(L', \Psi) \geq sup(L', \Psi)$. Putting all three inequalities together we get $\sigma > sup(L', \Psi)$, i.e., the antecedent. □

This result leads us to the following filter and refine strategy. Similar to the candidate generation step of the Apriori algorithm, location sets of increasing cardinality are constructed. Then, the weak support of the set is counted, and if this is below the threshold, the set is filtered out. At the end of entire process (when set cardinality reaches m), the refinement step is perfomed by explicitly counting the support of all surviving location sets.

Still, this approach could be inefficient, producing many false positives. It is possible that the support of a location set is below the threshold even though its weak support is above the threshold. Its support may even be zero if there exists no user that has posts covering all keywords. Such a location set cannot be pruned by Theorem 2. Following our example, consider location set $L = \{\ell_1, \ell_2\}$, keyword set $\Psi = \{\psi_1, \psi_2\}$, and assume that only user u_2 exists. In this case, $w\_sup(L, \Psi) = 1$, but $sup(L, \Psi) = 0$, since there exists no post from u_2 relevant to ψ_2. Motivated by this, we seek additional ways to identify location sets that cannot have high support. We first define the notion of a relevant user.

Definition 8 (Relevant User). *We say that a user u is* relevant *to a given keyword set Ψ, and denote as $u \in \mathcal{U}_{\Psi}$, if for each keyword $\psi \in \Psi$, the user has made a post relevant to ψ, i.e., the Association Graph contains an edge that is adjacent to ψ and includes u in its label.*

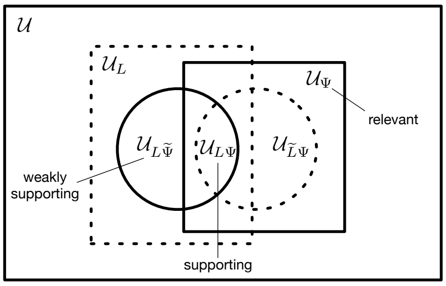


Figure 4: Set relationships between supporting, weakly supporting, and relevant users with respect to the association between location set L and keyword set Ψ.

Algorithm 1: Algorithm STA

```
   Input: keyword set $\Psi$, maximum cardinality $m$, support threshold $\sigma$
   Output: result set $\mathcal{R}^\sigma$ of all location sets with support at least $\sigma$
1  $\mathcal{R}^\sigma \leftarrow \emptyset$
2  $\mathcal{C}_1 \leftarrow \mathcal{L}$                                   ▷ candidate 1-location sets
3  $\mathcal{U}_\Psi \leftarrow$ IdentifyRelevantUsers($\Psi$)
4  for $1 \leq i \leq m$ do
5      $\mathcal{F}_i \leftarrow \emptyset$  ▷ $i$-location sets with more than $\sigma$ relevant and weakly supporting users
6      foreach $L \in \mathcal{C}_i$ do
7          ComputeSupports($L, \Psi$)
8          if $rw\_sup(L, \Psi) \geq \sigma$ then
9              $\mathcal{F}_i \leftarrow \mathcal{F}_i \cup \{L\}$
10             if $sup(L, \Psi) \geq \sigma$ then
11                 $\mathcal{R}^\sigma \leftarrow \mathcal{R}^\sigma \cup \{L\}$
12     $\mathcal{C}_{i+1} \leftarrow$ CandidateGeneration($\mathcal{F}_i$)   ▷ candidate $(i+1)$-location sets
```

Algorithm 2: STA.IdentifyRelevantUsers

```
   Input: keyword set $\Psi$
   Output: set $\mathcal{U}_\Psi$ of relevant users
1  $\mathcal{U}_\Psi \leftarrow \emptyset$
2  foreach $u \in \mathcal{U}$ do
3      $cov\Psi \leftarrow \emptyset$
4      foreach $p \in \mathcal{P}_u$ do
5          if $p.\psi \in \Psi$ then
6              $cov\Psi \leftarrow cov\Psi \cup \{\psi\}$
7      if $|cov\Psi| = |\Psi|$ then
8          $\mathcal{U}_\Psi \leftarrow \mathcal{U}_\Psi \cup \{u\}$
```

Notice that user u_2 is not relevant to $\Psi = \{\psi_1, \psi_2\}$. The next result shows that if we restrict the set of weakly supporting users to include only relevant users, we can still define a pruning rule.

Theorem 3. *If the number of relevant users that weakly support a location set L and a keyword set Ψ is less than σ, then the support of any location set $L' \supseteq L$ and Ψ cannot be more than σ.*

Proof. Recall that $\mathcal{U}_\Psi, \mathcal{U}_{L\widetilde{\Psi}}$ denote the set of relevant users and weakly supporting users, respectively. Then, the theorem assumes that $|\mathcal{U}_\Psi \cap \mathcal{U}_{L\widetilde{\Psi}}| < \sigma$. From (the proof of) Lemma 1 we have that $\mathcal{U}_{L\widetilde{\Psi}} \supseteq \mathcal{U}_{L'\widetilde{\Psi}}$. Therefore, $\mathcal{U}_\Psi \cap \mathcal{U}_{L\widetilde{\Psi}} \supseteq \mathcal{U}_\Psi \cap \mathcal{U}_{L'\widetilde{\Psi}}$ and thus $|\mathcal{U}_\Psi \cap \mathcal{U}_{L\widetilde{\Psi}}| \geq |\mathcal{U}_\Psi \cap \mathcal{U}_{L'\widetilde{\Psi}}|$. From (the proof of) Lemma 2 any user u that supports (L', Ψ) must also weakly support (L', Ψ). In addition, u must be relevant to Ψ due to the first condition of Definition 4. Hence, $|\mathcal{U}_\Psi \cap \mathcal{U}_{L'\widetilde{\Psi}}| \geq sup(L', \Psi)$. Combining the two derived inequalities and the theorem assumption, we derive that $sup(L', \Psi) < \sigma$. □

This result improves upon our filter and refine strategy, by allowing us to early prune a location set that cannot have support above σ, even though its weak support might be above σ.

A better way to understand the relation between the sets of supporting $\mathcal{U}_{L\Psi}$, weakly supporting $\mathcal{U}_{L\widetilde{\Psi}}$ and relevant $\mathcal{U}_\Psi$ users of a location set and keyword set (L, Ψ) is to draw a Venn diagram. Figure 4 depicts these sets, and also includes for completeness their dual sets drawn with dashed lines (discussed in Section 5.2). We have shown that while the cardinality of set $\mathcal{U}_{L\Psi}$ is not anti-monotone with respect to L, the cardinalities of sets $\mathcal{U}_{L\widetilde{\Psi}}$ and $\mathcal{U}_\Psi \cap \mathcal{U}_{L\widetilde{\Psi}}$ are. Figure 4 emphasizes that the intersection of relevant and weakly supporting users is a *tighter* superset of the desired supporting users set, while still allowing anti-monotonicity-based prunning. In the following, we write $rw\_sup(L, \Psi)$ to denote the number of relevant and weakly supporting users, i.e., $|\mathcal{U}_\Psi \cap \mathcal{U}_{L\widetilde{\Psi}}|$.

Returning to the example of Figure 2, the relevant to Ψ users are all except u_2. Therefore, we derive $sup(L, \Psi) = |\{u_1, u_3\}| = 2$, $w\_sup(L, \Psi) = |\{u_1, u_2, u_3\}| = 3$, and $rw\_sup(L, \Psi) = |\{u_1, u_3\}| = 2$, showing that the relevant and weak support is closer to the actual support than weak support is.

5. FINDING FREQUENT ASSOCIATIONS

We first present a baseline method for Problem 1, which serves as the foundation for more elaborate solutions based on indices.

5.1 Basic Algorithm

This algorithm implements the filter and refine approach discussed in Section 4. Recall that Theorems 2 and 3 allow to prune location sets with support less than σ based on the concepts of relevant and weakly supporting users (filter step). While this guarantees no false negatives, there can still be false positives, i.e., location sets with support less than σ, which need to be identified (refine step). Note that instead of performing this at the end, it can be done more efficiently during candidate generation, as explained later.

Algorithm 1 outlines the basic method, denoted as STA. It operates on the set $\mathcal{P}$ of posts organized by user, i.e., the list $\mathcal{P}_u$ containing the posts of each user u. The input includes the keyword set Ψ, the maximum cardinality m of a location set, and the support threshold σ. STA exploits the Apriori principle (lines 4–12) to identify the location sets with support above σ, filtering out each location set with fewer than σ relevant and weakly supporting users.

Initially, the result set is empty and the potential 1-location sets are set to all locations (lines 1–2). Also, the set of users relevant to Ψ is identified (line 3). Procedure IdentifyRelevantUsers, depicted in Algorithm 2, iterates across every list $\mathcal{P}_u$ and checks if user u has made posts that cover all keywords that appear in Ψ.

Then, STA proceeds in m iterations, following the Apriori principle. At the i-th iteration, all i-location sets with rw_sup not less than σ are stored in set $\mathcal{F}_i$. Among them, those with support not less than σ are added to the result set $\mathcal{R}^\sigma$. After initializing $\mathcal{F}_i$ (line 5), each candidate i-location set L is examined (lines 6–11). The set $\mathcal{C}_i$ of candidate i-location sets was generated at the end of the previous iteration (line 12) by the CandidateGeneration procedure that applies the Apriori principle. In particular, Candidate Generation creates candidate location sets of cardinality one more than what was just examined. It takes as input the i-location sets $\mathcal{F}_i$ with relevant weak support above σ, and inserts into $\mathcal{C}_{i+1}$ an $(i+1)$-location set only if all its i-location subsets are in $\mathcal{F}_i$, due to the Apriori principle implied by Theorem 3.

For candidate i-location set L, procedure ComputeSupports (described later) is invoked to determine the number $rw\_sup(L, \Psi)$ of relevant weakly supporting users, and the number $sup(L, \Psi)$ of supporting users (line 7). If the former support is above σ, L is added to $\mathcal{F}_i$ (lines 8–9). If, additionally, the latter support is greater than σ, then L is added to the result set $\mathcal{R}^\sigma$ (lines 10–11). This essentially corresponds to refining the surviving candidates.

Algorithm 3 depicts the pseudocode for ComputeSupports. The

Algorithm 3: STA.ComputeSupports

```
   Input: location set $L$, keyword set $\Psi$
   Output: weak support and support of $(L, \Psi)$
1  $r\_sup(L, \Psi) \leftarrow 0; sup(L, \Psi) \leftarrow 0$
2  foreach $u \in \mathcal{U}_\Psi$ do                                ▷ relevant user
3      $covL \leftarrow \emptyset; cov\Psi \leftarrow \emptyset$
4      foreach $p \in \mathcal{P}_u$ do
5          foreach $\ell \in L$ do
6              if $d(p.\ell, \ell) \leq \epsilon$ then
7                  foreach $\psi \in p.\Psi \cap \Psi$ do
8                      $covL \leftarrow covL \cup \{\ell\}$
9                      $cov\Psi \leftarrow cov\Psi \cup \{\psi\}$
10     if $|covL| = |L|$ then                            ▷ weakly supporting user
11         $rw\_sup(L, \Psi) \leftarrow rw\_sup(L, \Psi) + 1$
12         if $|cov\Psi| = |\Psi|$ then                  ▷ supporting user
13             $sup(L, \Psi) \leftarrow sup(L, \Psi) + 1$
```

Table 3: Support of Associations Between Listed Location Sets And Keyword Set $\Psi = \{\psi_1, \psi_2\}$ Based on the Posts in Figure 2

Location set	wr_sup	sup
$\{\ell_1\}$	3	1
$\{\ell_2\}$	3	1
$\{\ell_3\}$	3	0
$\boldsymbol{\{\ell_1, \ell_2\}}$	2	**2**
$\{\ell_1, \ell_3\}$	2	1
$\boldsymbol{\{\ell_2, \ell_3\}}$	3	**2**
$\{\ell_1, \ell_2, \ell_3\}$	1	1

procedure iterates over all relevant users. Let u be the currently examined user. The objective is to determine if u (weakly) supports (L, Ψ). For this purpose, the sets $covL$ and $cov\Psi$ are constructed to indicate what locations among L and keywords among Ψ, respectively, are covered by u. Each post of u is examined (lines 4–9). If the post's location is within distance ϵ to some location in $\ell \in L$, and there exists a keyword $\psi \in \Psi$ common with the post's keywords, then ℓ and ψ are inserted to $covL$ and $cov\Psi$ (lines 6–9). If all keywords in L have been found in u's relevant posts, then the counter of relevant and weakly supporting users is incremented (lines 10–11). Additionally, if all keywords appear in these posts, the counter for the support is incremented (lines 12–13).

Table 3 shows the relevant and weak support, and support for all location sets for keyword set $\Psi = \{\psi_1, \psi_2\}$, as computed by STA for the example of Figure 2 with support threshold $\sigma = 2$. Recall that all users except u_2 are relevant. As all 1-location sets have relevant and weak support above σ (although none is actually a result), all possible 2-location sets are constructed and their supports are counted. Among them, $\{\ell_1, \ell_2\}$ and $\{\ell_2, \ell_3\}$ (marked bold) have support 2 and are thus results. Observe the anti-monotonicity in relevant and weak support, and the lack thereof in support. Finally, as all 2-location sets have wr_sup above σ, the set $\{\ell_1, \ell_2, \ell_3\}$ is also considered but found to have low relevant and weak support.

5.2 Inverted Index-Based Algorithm

In STA, counting the weak support of a location set is particularly time consuming, since it scans the entire list of posts to find the weakly supporting users for each location. Even worse, if a location is part of multiple location sets, this is repeated multiple times.

To address this performance bottleneck, we present next an approach, termed STA-I, that is based on a preconstructed inverted index, which facilitates the identification of weakly supporting users for any location. The assumption here is that the distance parameter ϵ is known beforehand, i.e., it does not change with the queries.

Table 4: Inverted Index for the Posts in Figure 2

Location	Inverted list
ℓ_1	$\psi_1 : u_1, u_5,\ \psi_2 : u_3, u_5$
ℓ_2	$\psi_1 : u_1, u_3,\ \psi_2 : u_1, u_4$
ℓ_3	$\psi_1 : u_1, u_3, u_4$

Algorithm 4: STA-I.IdentifyRelevantUsers

```
Input: keyword set $\Psi$
Output: set $\mathcal{U}_\Psi$ of relevant users
$\mathcal{U}_\Psi \leftarrow \emptyset$
foreach $\psi \in \Psi$ do
    $\mathcal{C} \leftarrow \emptyset$
    foreach $\ell \in \mathcal{L}$ do
        $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{U}(\ell, \psi)$
    $\mathcal{U}_\Psi \leftarrow \mathcal{U}_\Psi \cap \mathcal{C}$
```

To construct the index, we identify the posts that are within distance ϵ to each location ℓ. Then, for each location, we compile an inverted list $\mathcal{U}(\ell)$, containing all users with posts local to ℓ. To further speed up processing, we partition each list according to keyword, such that each sublist $\mathcal{U}(\ell, \psi)$ contains users with posts local to ℓ and relevant to ψ. Table 4 shows the lists for our example. STA-I operates identically to STA, but uses the inverted index during the procedures IdentifyRelevantUsers and ComputeSupports.

The IdentifyRelevantUsers procedure is depicted in Algorithm 4. Recall, that the goal is to identify users who have made posts relevant to all keywords in Ψ, irrespective of the posts' geotags. Hence, for each keyword $\psi \in \Psi$, and each possible location ℓ, it retrieves the list $\mathcal{U}(\ell, \psi)$ of users with relevant and local posts, and it compiles the set of users with posts relevant to ψ and local to some location in L. Finally, it computes the intersection of these sets. This procedure essentially constructs the set of relevant users as $\mathcal{U}_\Psi = \bigcap_{\psi \in \Psi} \left(\bigcup_{\ell \in L} \mathcal{U}(\ell, \psi) \right)$.

Algorithm 5 illustrates the ComputeSupports procedure, which computes the weak support (lines 1–6) and the support (lines 8–14) of location set and keyword set (L, Ψ). Regarding the former, recall that a user weakly supports (L, Ψ) if for each location $\ell \in L$ there exists a local post that is relevant to some keyword in Ψ. The set $\bigcup_{\psi \in \Psi} \mathcal{U}(\ell, \psi)$ represents users that have relevant posts to some keyword in Ψ and are local to the specific location ℓ. Thus the intersection over all locations in L of these sets represents the weakly supporting users, i.e., $\mathcal{U}_{L\widetilde{\Psi}} = \bigcap_{\ell \in L} \left(\bigcup_{\psi \in \Psi} \mathcal{U}(\ell, \psi) \right)$. Specifically, the procedure computes the union in the inner loop (lines 3–4), and the intersection of the unions in the outer loop (lines 2–5). The weak support of (L, Ψ) is computed after the non-relevant users are discarded (line 6).

Only when the weak support of (L, Ψ) exceeds threshold σ (line 7), its support is computed (lines 8–14), but in a manner significantly different from that in STA. Recall from the discussion in Section 4 and Figure 4 that the set $\mathcal{U}_{L\widetilde{\Psi}}$ of weakly supporting users has a dual set $\mathcal{U}_{\widetilde{L}\Psi}$, termed local-weakly supporting users. This latter set contains users that for each keyword among Ψ have a post local to some location among L. It is not hard to see that the users that are both (relevant-)weakly supporting and local-weakly supporting (L, Ψ) are exactly those that support (L, Ψ), i.e., it holds that $\mathcal{U}_{L\Psi} = \mathcal{U}_{L\widetilde{\Psi}} \cap \mathcal{U}_{\widetilde{L}\Psi}$. Intuitively, the latter set satisfies the first requirement of Definition 4, whereas the former the second.

Based on this observation, the ComputeSupports procedure first computes the local-weakly supporting users $\mathcal{U}_{\widetilde{L}\Psi}$ (lines 8–13). With similar reasoning as before, the procedure builds the set as $\mathcal{U}_{\widetilde{L}\Psi} = \bigcap_{\psi \in \Psi} \left(\bigcup_{\ell \in L} \mathcal{U}(\ell, \psi) \right)$, where the union is compiled in the inner loop (lines 11–12), and the intersection of the unions in the outer loop (lines 9–13). Then, it intersects it with the previously constructed $\mathcal{U}_{L\widetilde{\Psi}}$ set to compute the support of (L, Ψ) (line 14).

Algorithm 5: STA-I.ComputeSupports

```
Input: location set $L$, keyword set $\Psi$
Output: weak support and support of $(L, \Psi)$
▷ construct set $\mathcal{U}_{L\widetilde{\Psi}}$ of (relevant-)weakly supporting users
$\mathcal{U}_{L\widetilde{\Psi}} \leftarrow \emptyset$
foreach $\ell \in L$ do
    $\mathcal{A} \leftarrow \emptyset$ foreach $\psi \in \Psi$ do
        $\mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{U}(\ell, \psi)$
    $\mathcal{U}_{L\widetilde{\Psi}} \leftarrow \mathcal{U}_{L\widetilde{\Psi}} \cap \mathcal{A}$
$rw\_sup(L, \Psi) \leftarrow |\mathcal{U}_{L\widetilde{\Psi}} \cap \mathcal{U}_{\Psi}|$
if $rw\_sup(L, \Psi) < \sigma$ then return
▷ construct set $\mathcal{U}_{\widetilde{L}\Psi}$ of local-weakly supporting users
$\mathcal{U}_{\widetilde{L}\Psi} \leftarrow \emptyset$
foreach $\psi \in \Psi$ do
    $\mathcal{B} \leftarrow \emptyset$
    foreach $\ell \in L$ do
        $\mathcal{B} \leftarrow \mathcal{B} \cup \mathcal{U}(\ell, \psi)$
    $\mathcal{U}_{\widetilde{L}\Psi} \leftarrow \mathcal{U}_{\widetilde{L}\Psi} \cap \mathcal{B}$
$sup(L, \Psi) \leftarrow |\mathcal{U}_{L\widetilde{\Psi}} \cap \mathcal{U}_{\widetilde{L}\Psi}|$
```

5.3 Spatio-Textual Index-Based Algorithm

Although precomputing the inverted index reduces dramatically the cost of calculating the weak support of a location set, it cannot handle different values of the distance parameter ϵ. Next, we present an alternative approach to accelerating weak support calculations based on spatio-textual indices. Instead of relying on precomputed static lists, we dynamically compile the information needed from the index. We first present a generic approach that works with the majority of existing spatio-textual indices, and then we consider a particular index and propose further optimizations.

5.3.1 Generic Algorithm

We adapt the basic Apriori-like algorithm assuming the availability of a spatio-textual index which can process spatio-textual range queries with OR semantics. The latter specify a spatial range R and a set of keywords Ψ, and seek all spatio-textual objects whose location is inside R and contain at least one of the keywords in Ψ.

We next describe the STA-ST algorithm which operates on top of such a general-purpose spatio-textual index. It operates similarly to STA, with the difference that procedure ComputeSupports is implemented in an index-aware manner, as outlined in Algorithm 6. It first constructs the set $\mathcal{U}_{L\widetilde{\Psi}}$ of weakly supporting users, and then determines the support of (L, Ψ). To build $\mathcal{U}_{L\widetilde{\Psi}}$, it issues a spatio-textual range query with parameters the disc (ℓ, ϵ) of radius ϵ around each location $\ell \in L$ and keyword set Ψ (lines 2–9). For a specific location ℓ, the results (set of posts) are stored in $\mathcal{P}_\ell$ (line 4). Then, it scans the results and inserts into a temporary variable $\mathcal{A}$ each encountered user $p.u$ (line 8). In addition, it associates with each user a bitmap $p.u.cov\Psi$ indicating which query keywords appear in her posts (lines 6–7); this information is later used to determine if the user supports (L, Ψ). Once all users with posts local to ℓ and relevant to Ψ have been identified in $\mathcal{A}$, they are merged with the ones for previously examined locations (line 9). Eventually, $\mathcal{U}_{L\widetilde{\Psi}}$ contains users with posts local to *every* location in L and relevant to at least one keyword in Ψ, i.e., the users weakly supporting (L, Ψ).

To compute the weak support among relevant users, the procedure takes the intersection of $\mathcal{U}_{L\widetilde{\Psi}}$ with the known set $\mathcal{U}_{\Psi}$ of relevant users (line 10). If the weak support is lower than the threshold, the algorithm returns (line 11). Otherwise it computes the support by examining whether each user has covered all query keywords (lines 13–15); this is determined directly from bitmaps $p.u.cov\Psi$.

5.3.2 Optimized Algorithm

Next, we focus on a specific spatio-textual index, I^3 [22], which we adapt to devise an even more efficient algorithm.

Algorithm 6: STA-ST.ComputeSupports

```
   Input: location set $L$, keyword set $\Psi$
   Output: weak support and support of $(L, \Psi)$
1  $\mathcal{U}_{L\widetilde{\Psi}} \leftarrow \emptyset$
2  foreach $\ell \in L$ do
3      $\mathcal{A} \leftarrow \emptyset$
4      $\mathcal{P}_\ell \leftarrow$ ST-RANGE$((\ell, \epsilon), \Psi)$
5      foreach $p \in \mathcal{P}_\ell$ do
6          foreach $\psi \in p.\Psi \cap \Psi$ do
7              $p.u.cov\Psi \leftarrow p.u.cov\Psi \cup \{\psi\}$
8          $\mathcal{A} \leftarrow \mathcal{A} \cup p.u$
9      $\mathcal{U}_{L\widetilde{\Psi}} \leftarrow \mathcal{U}_{L\widetilde{\Psi}} \cap \mathcal{A}$
10 $rw\_sup(L, \Psi) \leftarrow |\mathcal{U}_{L\widetilde{\Psi}} \cap \mathcal{U}_{\Psi}|$
11 if $rw\_sup(L, \Psi) < \sigma$ then return
12 $sup(L, \Psi) \leftarrow 0$
13 foreach $u \in \mathcal{U}_{L\widetilde{\Psi}}$ do
14     if $|u.cov\Psi| = |\Psi|$ then
15         $sup(L, \Psi) \leftarrow sup(L, \Psi) + 1$
```

We first elaborate on the index structure. For our purposes, the I^3 index can be seen as a quadtree that hierarchically partitions the spatial domain. Each node corresponds to a specific rectangular region, and points to its four children corresponding to the quadrants of the region. Leaf nodes point to disk pages containing the actual posts grouped by keyword. We associate with each node some additional aggregate information. Specifically, for each keyword ψ, we store the number of users with relevant posts that are contained within the subtree rooted at this node N. We denote this by $N.count(\psi)$.

STA-STO differs from STA-ST in the first iteration of the main Apriori loop (lines 4–12 of Algorithm 1 for $i = 1$). Instead of computing the weak support (and support) of every location, it uses the index to identify locations with potentially high weak support, eliminating groups of locations with weak support less than σ. To achieve this, it executes a best-first search (bfs) traversal [9], performing a simple test at each node to decide whether to continue in its subtree. Intuitively, we wish to terminate bfs when no location in the subtree can have weak support greater than σ.

Let Q be the priority queue implementing bfs. For each node N entering Q, the algorithm computes $a(N) = \sum_{\psi \in \Psi} N.count(\psi)$, and uses it as the queue's priority key. At each iteration, the node N in Q with the largest $a(N)$ value is removed. If $a(N)$ is greater than or equal to σ, there may exist some location in the subtree of N with weak support greater than σ. Otherwise, a safe conclusion cannot be drawn. Hence, the algorithm calculates an additional value $b(N)$ for this node, which is an upper bound on the weak support of any location within N. Clearly, if $b(N) < \sigma$, the node contains no useful locations and can be pruned. Such pruned nodes, along with their $a()$ values, are maintained in a deleted list D, which serves in the calculation of $b()$ values as explained next. For node N, its $b(N)$ value is the sum of $a()$ values for all nodes that are in Q or in D and that are within distance ϵ to N. An important observation here is that, due to the bfs traversal and the index structure, nodes in $Q \cup D$ do not spatially overlap and hence $b(N)$ does not double count posts. To summarize, STA-STO first makes the quick $a(N) \geq \sigma$ test, and only if this fails does it compute $b(N)$ and makes the more expensive $b(N) \geq \sigma$ test. If the latter fails too, the node definitely cannot contain a location set with weak support greater than σ.

For each location dequeued in the bfs traversal, STA-STO invokes the STA-ST.ComputeSupport procedure as described in the previous section, to determine its exact weak support and its support. Compared to it, the benefit is that STA-STO executes the procedure only for promising locations instead of every possible location.

6. FINDING TOP-K ASSOCIATIONS

Next, we present algorithms for Problem 2. We start with a basic

Algorithm 7: Algorithm κ-STA

```
Input: keyword set $\Psi$, maximum cardinality $m$, number of results $k$
Output: result set $\mathcal{R}^k$ containing top-$k$ location sets with highest support
$\sigma \leftarrow$ DetermineSupportThreshold($\Psi$, $k$)
$\mathcal{R}^\sigma \leftarrow$ STA ($\Psi$, $m$, $\sigma$)
$\mathcal{R}^k \leftarrow k$ location sets from $\mathcal{R}^\sigma$ with highest support
```

approach, and then discuss more efficient index-based techniques.

6.1 Basic Algorithm

In Problem 2, we seek the top-k location sets with the highest support, instead of setting a specific support threshold. However, a support threshold is needed in order to apply an Apriori-like method; thus, we explain how such a threshold can be computed. If we pick any set of k distinct location sets and compute their supports, then the minimum value among those can serve as the support threshold σ; clearly, any other set with support lower than this cannot be in the result. The challenge is to construct initial location sets with high support so that the starting value of σ is effectively high.

Algorithm 7 outlines the generic method κ-STA implementing this simple idea. First, procedure DetermineSupportThreshold is invoked to obtain an appropriate lower bound σ on the support of the top-k set. Given σ, it invokes the STA algorithm to derive all location sets with support above σ. Finally, among the returned location sets, it returns the k with the highest support.

Regarding the DetermineSupportThreshold procedure, the main idea is to construct at least k distinct location sets that cover all keywords Ψ. Suppose that for each keyword $\psi \in \Psi$ we have determined $k(\psi)$ distinct locations with local posts relevant to ψ. Combining these $k(\psi)$ distinct locations for each keyword, we can construct distinct location sets. Note that a necessary condition to obtain k location sets is $\prod_{\psi \in \Psi} k(\psi) \geq k$.

Following this process, a heuristic for obtaining combinations with high support is to start with locations that are popular, i.e., have high weak support. In the absence of any index, procedure DetermineSupportThreshold iterates over the set of posts lists $\mathcal{P}_u$, skipping users that do not have relevant posts to each ψ. For the rest, the locations of the relevant posts to each ψ are noted. In addition, a counter for the weak support of each location is maintained. After a sufficient number of locations for each keyword are seen, the procedure terminates. For each keyword, the locations with the highest weak support are chosen and combined. The support of each set is computed by ComputeSupports, and the k-th highest among these values is set as the support threshold σ.

6.2 Index-Based Algorithms

6.2.1 Inverted Index

When an inverted index from locations to users with local posts is available, DetermineSupportThreshold collects locations with local posts relevant to each keyword in Ψ in a different manner. It first computes the weak support of every location by invoking ComputeSupports. Note that this has to be executed anyway when we later invoke the STA-I algorithm irrespective of the support threshold σ. Then, it examines locations in descending order of their weak support. For each location ℓ, the procedure checks the inverted list and associates the location with each keyword in Ψ for which a local and relevant post exists. Similar to the basic algorithm, once a sufficient number of locations per keyword are seen, location sets are generated and their support is computed.

6.2.2 Spatio-Textual Index

In a generic spatio-textual index, DetermineSupportThreshold operates identically to the basic algorithm with the exception that the ComputeSupports procedure is index-aware. When the augmented I^3 index is used, a different process is followed. Procedure DetermineSupportThreshold first performs a best-first search traversal similar to that described in Section 5.3.2. The difference is that initially there is no support threshold, and thus the $b()$ values need not be computed. Moreover, the traversal is progressive, meaning that at each step the next location with potentially high weak support is identified. For each such location, its local posts are retrieved (using the index) and it is marked for the keywords that appear in these posts. As before, once a sufficient number of locations per keyword are seen, the support threshold is computed.

Table 5: Dataset Characteristics

Dataset	Num. of photos	Num. of users	Num. of distinct tags	Avg. num. of tags per photo	Avg. num. of tags per user	Num. of locations
London	1,129,927	16,171	266,495	8.1	61.2	48,547
Berlin	275,285	7,044	88,783	8.1	39.4	21,427
Paris	549,484	11,776	122,998	7.8	38.8	38,358

7. EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation of our approach using real-world datasets comprising geolocated Flickr photos. We first describe our experimental setup, outlining the datasets and the queries used in the experiments, and then we report and discuss the results.

7.1 Datasets

In our experiments, we have used geolocated photos from Flickr, extracted from a large-scale dataset that is provided publicly by Yahoo! for research purposes [18]. Specifically, we compiled datasets for the cities of London, Berlin and Paris. For each dataset, Table 5 lists the number of photos, users, and distinct keywords contained in it, as well as the average number of keywords per photo and distinct keywords per user. As a database of locations, we used POIs collected from the Foursquare API[1]. The number of distinct locations per city is also shown in Table 5.

To construct a keyword set that is used to search for socio-textual associations, we followed the process described next. First, for each dataset, we retrieved the 100 most frequent keywords, where the frequency of a keyword was measured by the number of users having photos with it. From those, we manually picked a set of 30 keywords, removing more generic ones, such as *"london"*, *"england"*, *"uk"*, *"iphone"*, *"canon"*, etc. The top 10 selected keywords for each city are listed in Table 6, showing also the number of users with relevant posts to each one. Then, we combined these popular keywords to create keyword sets of cardinality up to 4. For each case, we selected the top 20 combinations according to the number of users having photos with those tags. Table 7 lists the first 5 among these 20 combinations for each case. In all reported experiments, we set the value of the spatial distance threshold parameter ϵ, used to associate photos to locations, to 100 meters.

7.2 Indicative Result

Figure 5 shows an example of the socio-textual associations our methodology discovers. In particular, we look for locations that are strongly associated with the keyword set $\Psi = \{$*"london eye"*, *"thames"*$\}$. The depicted green (resp., purple) points denote the locations of photos that contain the keyword *"thames"* (resp., *"london eye"*) and belong to relevant users, i.e., they have also posted photos containing the other keyword *"london eye"* (resp., *"thames"*).

[1] https://developer.foursquare.com/

Table 6: Most Popular Keywords

London	Berlin	Paris
thames (2752)	reichstag (876)	louvre (2287)
park (1738)	fernsehturm (774)	eiffel+tower (1742)
london+eye (1730)	architecture (716)	seine (1488)
big+ben (1698)	alexanderplatz (713)	notre+dame (1244)
westminster (1543)	wall (684)	street (1194)
architecture (1519)	graffiti (575)	montmartre (1184)
museum (1386)	street (562)	architecture (1136)
art (1319)	art (543)	museum (1022)
tower+bridge (1276)	museum (526)	church (980)
statue (1178)	spree (492)	art (970)

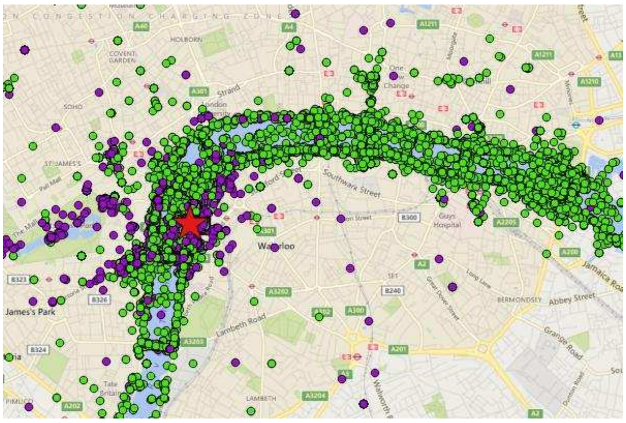


Figure 5: Indicative example for London with $\Psi = \{london\ eye, thames\}$.

We can see that photos about *"thames"* are spread along the entire length of the river Thames. On the other hand, although London Eye has a specific location, due to its high visibility relevant photos can be found at various other locations, e.g., in and around St. James Park. Nevertheless, since London Eye happens to be located at the bank of river Thames, the regions covered by the respective sets of relevant photos have a high overlap. In fact, the single location drawn as a star in this overlap has the strongest association with the keyword set. In this case, there exists a singleton location set that covers both keywords and has the highest support in the data.

7.3 Comparison with Other Association Types

As already explained (see Sections 1 and 2), there exist various approaches that discover different associations between locations and a given set of keywords. Hence, the purpose of our next experiment was to investigate whether the location sets returned by our approach (STA) are significantly different from those returned by other works, collective spatial keywords (CSK) and aggregate popularity (AP). We note that we cannot compare with approaches that discover location patterns (LP) as they ignore textual information.

To that end, we computed the top 10 results for STA, AP, and CSK, with respect to the keyword sets we compiled for the three datasets of London, Berlin, and Paris. Then, we computed the Jaccard similarity of the result sets of CSK and AP to ours. This measures the overlap in the query results, i.e., how many location sets STA and either CSK or AP return in common.

The results of this experiment are presented in Table 8. The results are averaged across queries with the same keyword set cardinality. As can be observed, the Jaccard similarity scores are very low in all cases, with values not exceeding 0.3. The highest scores are observed for queries with 2 keywords, where fewer possible location combinations exist. In those queries, on average, around 2 or 3 of the top 10 location sets discovered by STA are common with those appearing in the results of AP or CSK. The degree of overlap drops even lower when the cardinality of the keyword set increases, allowing for a significantly larger number of candidate location sets. In those cases, often there is only one or zero results in common. This outcome is consistent across the three datasets.

These results show that STA constitutes a novel and distinct criterion for discovering interesting socio-textual associations among locations, which cannot be replicated by existing approaches.

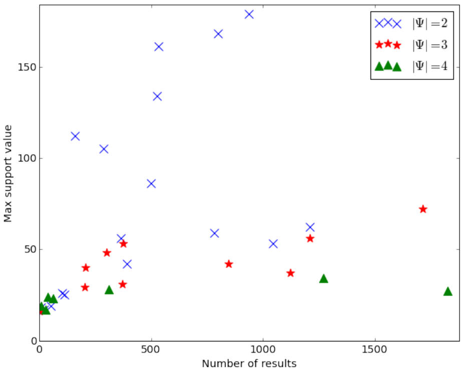


Figure 6: Scatter plot where data points correspond to experiments with distinct keyword sets; the x axis indicates the number of associations above the support threshold, and the y axis indicates the highest support among the associations.

7.4 Number of Discovered Associations and Maximum Support

Another aspect to investigate is the distribution of the number of results (associations found) and the support scores for different keyword set cardinalities. To that end, we computed the results for all keyword sets described in Section 7.1, i.e., 60 sets for each dataset, with cardinality $|\Psi| \in [2, 4]$. For each keyword set of the respective dataset, we measured the number of results and the support of the top result. The results of this experiment are shown in Figure 6. We only present results for London; the other two datasets exhibited a similar pattern, and are hence omitted. In this result, the support threshold parameter was set $\sigma = 0.1\%$ of the total number of users in the London dataset. Note that the value of the support threshold affects both the execution time and the number of results to be found. On the one hand, if the threshold is set too low, an excessive number of results may be returned, and the execution time may also be too high, since only few combinations can be pruned; on the other hand, setting the support threshold too high may return no results. Thus, the above value was selected experimentally according to this trade-off.

We notice the following trend in the results. Having only two keywords tends to produce results with high support (e.g., up to around 3% of the total number of users). As the number of keywords increases to 3 or 4, the maximum support among the returned results reduces significantly, dropping close to the support threshold; however, the number of returned results becomes much higher. This is an effect of the fact that, as explained in Section 4, the anti-monotonicity property does not hold in our problem.

7.5 Evaluation Time

Finally, we evaluate the efficiency of our proposed algorithms. In this experiment, we used the same keyword sets as above.

First, we compare the execution time of the three algorithms, `STA-I`, `STA-ST` and `STA-STO`, while varying the support threshold

Table 7: Most Popular Keyword Sets

$\lvert\Psi\rvert$	**London**
2	london+eye, thames (922); big+ben, london+eye (908); thames, westminster (898); park, thames (880); big+ben, thames (846)
3	big+ben, london+eye, thames (557); big+ben, thames, westminster (497); big+ben, london+eye, westminster (472); london+eye, thames, westminster (464); park, thames, westminster (440)
4	big+ben, london+eye, thames, westminster (358); big+ben, london+eye, thames, tower+bridge (293); art, green, park, thames (258); green, park, thames, trees (257); park, statue, thames, westminster (257)
	Berlin
2	alexanderplatz, fernsehturm (404); fernsehturm, reichstag (320); alexanderplatz, reichstag (253); reichstag, wall (249); fernsehturm, spree (248)
3	alexanderplatz, fernsehturm, reichstag (192); alexanderplatz, fernsehturm, spree (166); alexanderplatz, fernsehturm, wall (145); brandenburger+tor, fernsehturm, reichstag (144); fernsehturm, reichstag, spree (142)
4	alexanderplatz, fernsehturm, reichstag, spree (106); alexanderplatz, brandenburger+tor, fernsehturm, reichstag (96); alexanderplatz, fernsehturm, reichstag, wall (95); alexanderplatz, fernsehturm, potsdamer+platz, reichstag (90); alexanderplatz, fernsehturm, museum, reichstag (82)
	Paris
2	eiffel+tower, louvre (777); louvre, seine (745); louvre, museum (706); louvre, notre+dame (691); eiffel+tower, notre+dame (606)
3	eiffel+tower, louvre, notre+dame (415); eiffel+tower, louvre, seine (343); louvre, notre+dame, seine (339); louvre, river, seine (327); arc+de+triomphe, eiffel+tower, louvre (324)
4	eiffel+tower, louvre, notre+dame, seine (215); bridge, louvre, river, seine (209); arc+de+triomphe, eiffel+tower, louvre, notre+dame (208); louvre, museum, river, seine (189); bridge, river, seine, street (187)

Table 8: Degree of Overlap Between the Associations Discovered by STA and those of Existing Approaches

	London		**Berlin**		**Paris**	
$\lvert\Psi\rvert$	**AP**	**CSK**	**AP**	**CSK**	**AP**	**CSK**
2	0.22	0.24	0.28	0.30	0.20	0.14
3	0.17	0.04	0.09	0.07	0.08	0.03
4	0.14	0.03	0.01	0.04	0.00	0.00

Table 9: Ratio of Number of Location Sets with Support Above Threshold over Number of Location Sets with Weak Support Above Threshold; $\sigma = 0.2\%$

$\lvert\Psi\rvert$	**London**	**Berlin**	**Paris**
2	13.29%	23.80%	25.98%
3	1.35%	1.09%	3.85%
4	0.01%	0.00%	0.36%

parameter σ, which is a percentage of the number of users in each dataset. Note that the basic STA method was at least an order of magnitude slower than all other methods and is thus omitted from all plots. Moreover, we include STA-ST in the comparison, in order to assess the benefits resulting by the STA-STO optimizations. The results are presented in Figures 7 and 8, for 2 and 4 keywords, respectively; results for $|\Psi| = 3$ are similar and are omitted.

As the support threshold increases, the performance of all methods improves because fewer location sets survive the pruning. This is apparent in Paris, but not so much in London and Berlin for the specific range of support values depicted. Clearly, STA-I achieves the best performance. This is not surprising, since exploiting the preconstructed inverted index saves a substantial amount of the execution time during evaluation. It is worth noticing, however, that STA-STO is also very efficient, achieving competitive execution times compared to STA-I. In fact, this is not a merit of the spatio-textual index per se, but rather a result of the proposed optimizations; indeed, the execution times of the generic STA-ST are higher by an order of magnitude. The results appear to be consistent across the different datasets and for different number of keywords.

Table 9 quantifies the number of location sets (or associations) discovered that have weak support above but actual support below the threshold, which was set to $\sigma = 0.2\%$. For example, in London for $\Psi = 2$, we have that 13.29% of the location sets considered are actual results. As the keyword cardinality increases, the ratio decreases dramatically, because it becomes harder for location sets with weak support above the threshold to also cover all keywords.

Finally, we evaluate the performance of the algorithms for the top-k version of the problem. The results are presented in Figure 9 for $|\Psi| = 3$. A similar outcome is observed, with K-STA-I outperforming K-STA-STO in all cases. For both algorithms, the execution time tends to increase with k as more results are requested.

8. CONCLUSIONS

In this paper, we have addressed the problem of finding socially and textually associated location sets from user trails on the Web. We have formally defined the problem and studied its characteristics. Based on this, we have proposed a general approach for addressing the problem, which we have elaborated to derive three algorithms based on different indices. Furthermore, we have extended our approach to address also the top-k variant of the problem. The proposed methods have been evaluated experimentally using geotagged Flickr photos in three different cities.

Acknowledgements

This work was partially supported by the EU Project City.Risks (H2020-FCT-2014-653747).

9. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.

[2] M. Becker, P. Singer, F. Lemmerich, A. Hotho, D. Helic, and M. Strohmaier. Photowalking the city: Comparing hypotheses about urban photo trails on Flickr. In *SocInfo*, pages 227–244, 2015.

[3] G. Cai, C. Hio, L. Bermingham, K. Lee, and I. Lee. Mining frequent trajectory patterns and regions-of-interest from Flickr photos. In *HICSS*, pages 1454–1463, 2014.

[4] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD*, pages 373–384, 2011.

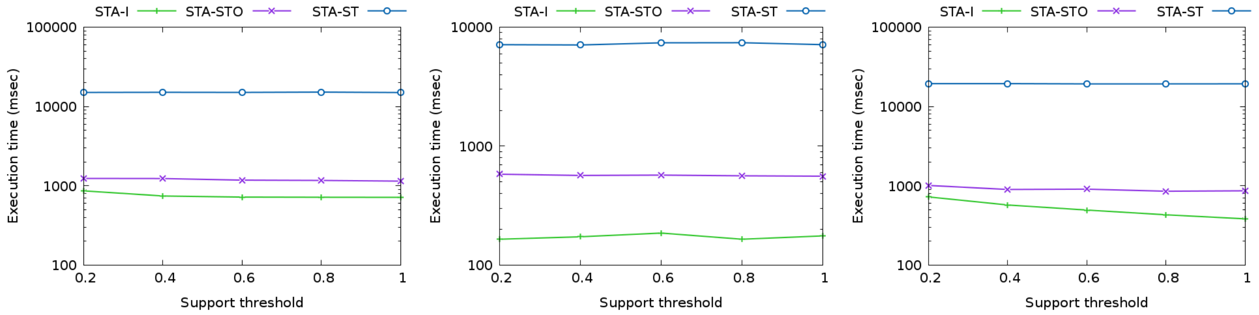
[5] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*, 6(3):217–228, 2013.

[6] M. D. Choudhury, M. Feldman, S. Amer-Yahia, N. Golbandi, R. Lempel, and C. Yu. Automatic construction of travel itineraries using social breadcrumbs. In *HT*, pages 35–44, 2010.

[7] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel. Text vs. space: efficient geo-search query processing. In *CIKM*, pages 423–432, 2011.

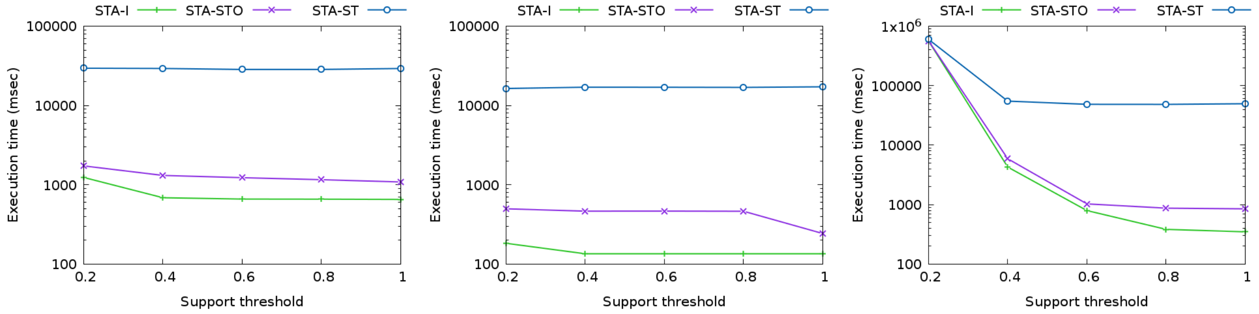
[8] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *WWW*, pages 613–622, 2001.

[9] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM TODS*, 24(2):265–318, 1999.



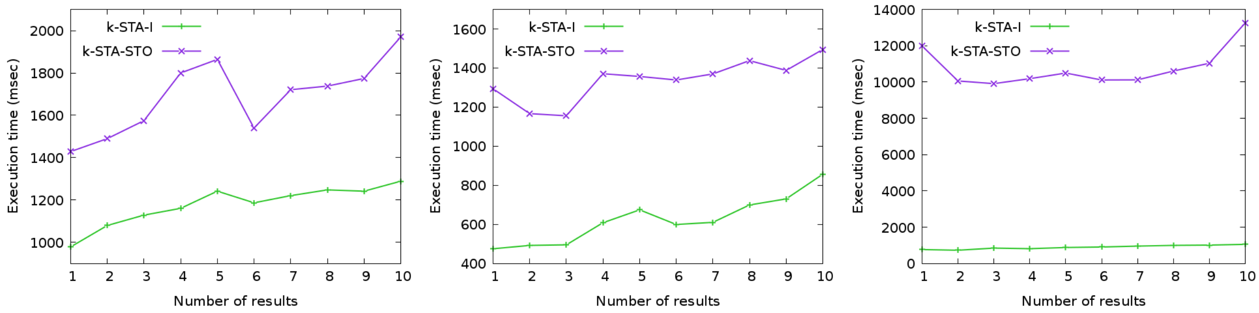
(a) London (b) Berlin (c) Paris

Figure 7: Varying support threshold (σ); $|\Psi| = 2$.



(a) London (b) Berlin (c) Paris

Figure 8: Varying support threshold (σ); $|\Psi| = 4$.



(a) London (b) Berlin (c) Paris

Figure 9: Varying number of results (k); $|\Psi| = 3$.

[10] S. Kisilevich, D. A. Keim, and L. Rokach. A novel approach to mining travel sequences using collections of geotagged photos. In *AGILE*, pages 163–182, 2010.

[11] T. Kurashima, T. Iwata, G. Irie, and K. Fujimura. Travel route recommendation using geotags in photo sharing sites. In *CIKM*, pages 579–588, 2010.

[12] I. Lee, G. Cai, and K. Lee. Mining points-of-interest association rules from geo-tagged photos. In *HICSS*, pages 1580–1588, 2013.

[13] X. Lu, C. Wang, J. Yang, Y. Pang, and L. Zhang. Photo2Trip: generating travel routes from geo-tagged photos for trip planning. In *Multimedia*, pages 143–152, 2010.

[14] A. Majid, L. Chen, G. Chen, H. T. Mirza, I. Hussain, and J. Woodward. A context-aware personalized travel recommendation system based on geotagged social media data mining. *International Journal of Geographical Information Science*, 27(4):662–684, 2013.

[15] E. Spyrou, I. Sofianos, and P. Mylonas. Mining tourist routes from flickr photos. In *SMAP*, pages 1–5, 2015.

[16] Y. Sun, H. Fan, M. Bakillah, and A. Zipf. Road-based travel recommendation using geo-tagged images. *Computers, Environment and Urban Systems*, 53:110–122, 2015.

[17] C. Tai, D. Yang, L. Lin, and M. Chen. Recommending personalized scenic itinerary with geo-tagged photos. In *ICME*, pages 1209–1212, 2008.

[18] B. Thomee, D. A. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L.-J. Li. The new data and new challenges in multimedia research. *arXiv preprint arXiv:1503.01817*, 2015.

[19] Z. Yin, L. Cao, J. Han, J. Luo, and T. S. Huang. Diversified trajectory pattern ranking in geo-tagged social media. In *SDM*, pages 980–991, 2011.

[20] D. Zhang, C. Chan, and K. Tan. Processing spatial keyword query as a top-k aggregation query. In *SIGIR*, pages 355–364, 2014.

[21] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *ICDE*, pages 688–699, 2009.

[22] D. Zhang, K.-L. Tan, and A. K. Tung. Scalable top-k spatial keyword search. In *EDBT*, pages 359–370, 2013.

[23] Y. Zheng, Z. Zha, and T. Chua. Mining travel patterns from geotagged photos. *ACM TIST*, 3(3):56, 2012.

[24] Y. Zheng, L. Zhang, X. Xie, and W. Ma. Mining interesting locations and travel sequences from GPS trajectories. In *WWW*, pages 791–800, 2009.

[25] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W. Ma. Hybrid index structures for location-based web search. In *CIKM*, pages 155–162, 2005.

COP: Planning Conflicts for Faster Parallel Transactional Machine Learning

Faisal Nawab[1] Divyakant Agrawal[1] Amr El Abbadi[1] Sanjay Chawla[2,3]

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

[2]Qatar Computing Research Institute, HBKU
schawla@qf.org.qa
[3]University of Sydney, Sydney, NSW, Australia
sanjay.chawla@sydney.edu.au

ABSTRACT

Machine learning techniques are essential to extracting knowledge from data. The volume of data encourages the use of parallelization techniques to extract knowledge faster. However, schemes to parallelize machine learning tasks face the trade-off between obeying strict consistency constraints and performance. Existing consistency schemes require expensive coordination between worker threads to detect conflicts, leading to poor performance. In this work, we consider the problem of improving the performance of multi-core machine learning while preserving strong consistency guarantees.

We propose Conflict Order Planning (COP), a consistency scheme that exploits special properties of machine learning workloads to reduce the overhead of coordination. What is special about machine learning workloads is that the dataset is often known prior to the execution of the machine learning algorithm and is reused multiple times with different settings. We exploit this prior knowledge of the dataset to plan a partial order for concurrent execution. This planning reduces the cost of consistency significantly because it allows the use of a light-weight conflict detection operation that we call ReadWait. We demonstrate the use of COP on a Stochastic Gradient Descent algorithm for Support Vector Machines and observe better scalability and a speedup factor between 2-6x when compared to other consistency schemes.

1. INTRODUCTION

The increasingly larger sizes of machine learning datasets have motivated the study of scalable parallel and distributed machine learning algorithms [7, 16, 20–22, 24, 25, 27]. The key to a scalable computation is the efficient management of coordination between processing workers, or workers for short. Some machine learning algorithms require only a small amount of coordination between workers making them easily scalable. However, the vast majority of machine learning algorithms are studied and developed in the serial setting, which makes it arduous to distribute these *serial-based algorithms* while maintaining the algorithm's behavior and goals.

Distributing serial-based algorithms may be performed by encapsulating the algorithm within existing parallel and distributed computation frameworks. These frameworks are oblivious to the actual computation. Thus, the machine learning algorithms may be incorporated as-is without redesign. In this paper, we consider a framework of *transactions* [3, 10] for parallel multi-core execution of machine learning algorithms. A transaction may represent the processing of an iteration of the machine learning algorithm where workers run transactions in parallel. *Serializability* is the correctness criterion for transactions that ensures that the outcome of a parallel computation is equivalent to some serial execution. To guarantee serializability, transactions need to coordinate via consistency schemes such as locking [8] and optimistic concurrency control (OCC) [15].

Recently, coordination-free approaches to parallelizing machine learning algorithms have been proposed [7, 24, 25]. In these approaches, workers do not coordinate with each other thus improving performance significantly compared to methods like locking and OCC. Although these techniques were very successful for many machine learning problems, there is a concern that the coordination-free approach leads to "requiring potentially complex analysis to prove [parallel] algorithm correctness" [21]. When a machine learning algorithm, $\mathbb{A}$, is developed, it is accompanied by mathematical proofs to verify its theoretical properties, such as convergence. These proofs are typically on the serial-based algorithm. A coordination-free parallelization of a proven serial algorithm, denoted $\varphi_{cf}(\mathbb{A})$, is not guaranteed to have the same theoretical properties as the serial algorithm $\mathbb{A}$. This is due to overwrites and inconsistency that makes the outcome of $\varphi_{cf}(\mathbb{A})$ different from $\mathbb{A}$. Thus, guaranteeing the theoretical properties requires a separate mathematical analysis of $\varphi_{cf}(\mathbb{A})$, that although possible [6, 25], can be complex. Additionally, the theoretical analysis might reveal the need for changes to the algorithm to preserve its theoretical guarantees in the parallel setting [25].

Running parallel machine learning algorithms in a serializable, transactional framework bypasses the need for an additional theoretical analysis of the correctness of parallelization. This is because a serializable parallel execution, denoted $\varphi_{SR}(\mathbb{A})$, is equivalent to some serial execution of $\mathbb{A}$, and thus preserves its theoretical properties. We will call parallelizing with serializability, the *universal approach* because serial machine learning algorithms are applied to it

Series ISSN: 2367-2005

10.5441/002/edbt.2017.13

without the need of additional theoretical analysis or changes to the original algorithm.

In this work, we focus on the universal approach of parallelizing machine learning algorithms with serializable transactions. Consistency schemes like locking [8, 11], OCC [25], and others [2] incur a significant performance overhead. Traditional serializability schemes were designed mainly for database workloads. Database workloads are typically arbitrary, unrepeatable units of work that are unknown to the database engine prior to execution. This is not the case for machine learning workloads. Machine learning tasks are well defined. Most machine learning algorithms apply a single iterative task repeatedly to the dataset. Also, the dataset (*i.e.*, the machine learning workload) is typically processed multiple times within the same run of the algorithm, and is potentially used for different runs with different machine learning algorithms. Generally, machine learning datasets are also known, in offline settings, prior to the experiments. These properties of machine learning workloads make it feasible to plan execution. We call these the *dataset knowledge* properties.

We propose Conflict Order Planning (COP) for parallel machine learning algorithms. COP ensures a serializable execution that preserves the theoretical guarantees of the serial machine learning algorithm. It leverages the dataset knowledge properties of machine learning workloads to plan a partial order for concurrent execution that is serializable. It annotates each transaction (*i.e.*, a machine learning iteration) with information about its dependencies according to the planned partial order. At execution time, these *planned dependencies* must be enforced. Enforcing a planned dependency is done by validating that an operation reads or overwrites the correct version according to the plan. This validation is done using a light-weight operation that we call ReadWait. This operation is essentially an arithmetic operation that compares version numbers, which is a much lighter operation compared to locking and other traditional consistency schemes.

We present background about the problem, the system and transactional machine learning model in Section 2. Then, we propose COP in Section 3 followed by correctness proofs in Section 4. We present our evaluation in Section 5. The paper concludes with a discussion of related work and a conclusion in Sections 6 and 7.

2. BACKGROUND

In this section, we provide the necessary background for the rest of this paper. We introduce use cases of planning within machine learning systems in Section 2.1. Section 2.2 presents the transactional model we will use for machine learning algorithms.

2.1 Use Cases

We now demonstrate the opportunity and rewards of planning machine learning execution in three common models of machine learning systems. We revisit these use cases in the paper when appropriate to show how COP planning applies to them.

2.1.1 Machine Learning Framework

Machine learning and data scientists do not process a dataset only once in their process of analyzing it. Rather, the scientist works on a dataset continuously, experimenting with different methods and machine learning algorithms to discover what method works best with a dataset. Thus, the same dataset is being processed by many machine learning algorithms repeatedly. Figure 1 shows a typical flow diagram of a machine learning framework [12, 14, 27]. Multiple machine learning algorithms are applied to an input dataset to produce models of the dataset. Each machine learning algorithm may be applied multiple times with different configuration and parameters, such as the learning rate.

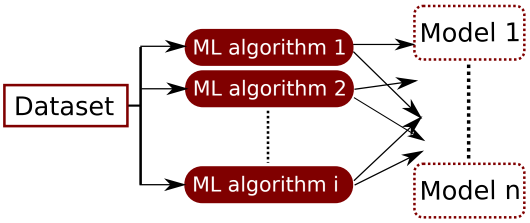


Figure 1: A flow diagram of a typical machine learning framework that employs a number of machine learning algorithms to learn models from a dataset

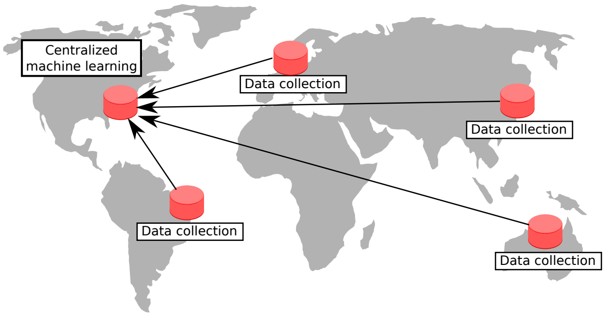


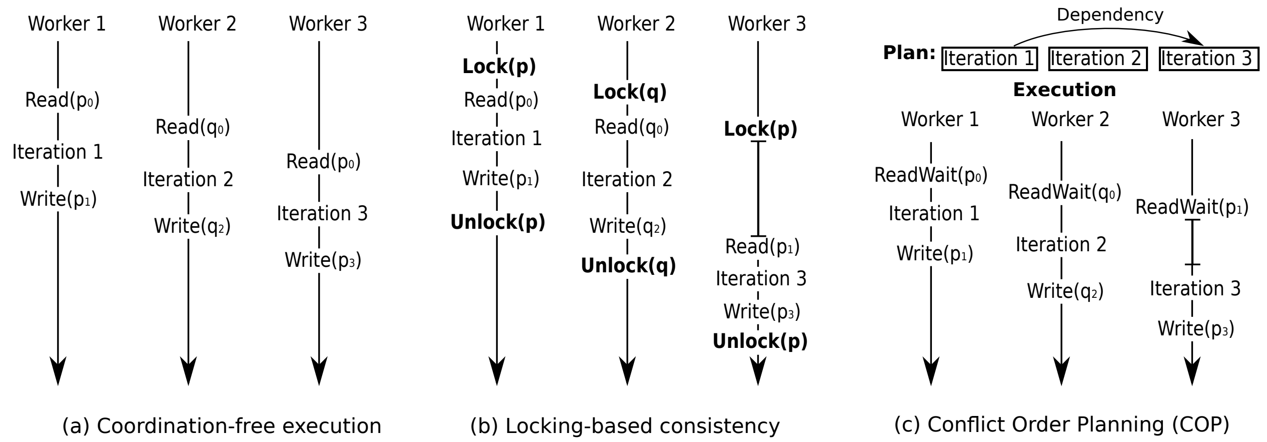
Figure 2: The current practice of machine learning of data collected across the world is to batch data at geo-distributed datacenters and send batches to a centralized location that performs the machine learning algorithm

In this model of a machine learning framework, the dataset is being processed many times, once for each generated model. This is an opportunity for COP to perform a planning stage that is then applied to all runs.

2.1.2 Global-Scale Machine Learning

Online machine learning is the practice of learning from data or events in real time. An example is a web application that collects user interactions with the website and generates a user behavior model using machine learning. Another example is applying machine learning to data collected by Internet of Things (IoT) and mobility devices. Typically, data is *born* around the world, collected at different datacenters, and then sent to a single datacenter that contains a centralized machine learning system. This case is shown in Figure 2 where there is a *central datacenter* for machine learning in North America and four other *collection datacenters* that collect and send data. This model has been reported to be the current standard practice of global-scale machine learning [4].

As data is being collected and batched at collection datacenters, there is an opportunity to generate a COP plan. This plan is then applied at the central datacenter for faster



(a) Coordination-free execution

(b) Locking-based consistency

(c) Conflict Order Planning (COP)

Figure 3: Execution of a machine learning algorithm by three workers with different consistency schemes. Each worker processes an iteration of the machine learning algorithm, where the first and third iterations read and update the same model parameter.

execution. A challenge in this model is that data is generated at different locations simultaneously and continuously. In such cases, COP plans for each batch individually at collection datacenters, and then batches are processed at the centralized datacenter in tandem.

2.1.3 Dataset Loading, Preprocessing and Execution

In addition to the opportunities for planning shown in use cases of machine learning systems, there is an opportunity for planning even in a single execution of a machine learning algorithm on a single dataset. This is because, typically, two tasks are performed prior to a machine learning algorithm execution: (1) Loading the dataset to main memory. Before execution, the dataset is stored in persistent storage, such as a disk. While loading the dataset from persistent storage, there is an opportunity to perform additional work to plan the execution. Our experiments demonstrate that planning while loading the dataset introduces a small overhead between 3% and 5% (Section 5.3).

Datasets are also typically preprocessed for various purposes such as formatting, data cleaning, and normalization [12]. Preprocessing is normally performed on the whole dataset, thus introducing an opportunity to plan execution while preprocessing is performed.

Even in the case of a dataset that is already preprocessed, loaded and ready to be learned, there is another opportunity to plan execution. A machine learning algorithm processes a dataset in multiple *rounds* on the dataset that we call epochs. Thus, planning during the first epoch will be rewarding for the execution of the remaining epochs.

In Section 3 we introduce COP planning algorithms and discuss their application to the various use cases we have presented.

2.2 Transactional Model of Machine Learning

A machine learning algorithm creates a mathematical model of a problem by iteratively learning from a dataset. The mathematical model of a machine learning algorithm is represented by *model parameters*, P, or parameters for short. For example, the mathematical model of linear regression takes the form $y = \sum_{i=1}^{n} \beta_i x_i + \epsilon$. The model parameters consist of the variables of the model, namely the vector of coefficients, β, and ϵ. The machine learning algorithm uses the dataset to estimate the parameter values that will result in the best fit to predict the dependent variable, y.

A dataset, $\mathbb{D}$, contains a number of samples, where the i^{th} sample is denoted $\mathbb{D}_i$. Each sample contains information about a subset of the parameters and the dependent variable corresponding to them. To distinguish between model parameters and parameter values in samples, we call the parameter values in samples *features*. For example, a dataset may contain information about movies. Each sample contains a list of the actors in a movie and whether the movie has a high rating. A mathematical model can be constructed to predict whether a movie has a high rating given the list of actors in it. Each parameter in the model corresponds to an actor. A sample contains a vector of feature values, where a feature has a value of 1 if the actor corresponding to it is part of the movie and 0 otherwise. A sample in the dataset also contains whether the movie has a high rating. Using the dataset, the mathematical model is constructed by estimating parameter values. These parameter values can then be utilized in the mathematical model to predict whether a new movie will have a high rating based on the actors in it.

Estimating model parameters is performed by iteratively learning from the dataset. Each iteration processes a single sample or a group of samples to have a better estimate of model parameters. An *epoch* is a collection of iterations that collectively process the whole dataset once. Machine learning algorithms run for many epochs until convergence. For example, Stochastic Gradient Descent (SGD) processes a single sample in each iteration. In an iteration, gradients are computed using a cost function to minimize the error in estimation. The gradients are then used to update the model parameters.

Machine learning algorithms are typically studied and designed for a serial execution where iterations are processed one iteration at a time. A straightforward approach to parallelizing a machine learning computation is to make workers process iterations concurrently, where each worker is responsible for the execution of a different iteration. Executing iterations concurrently may lead to conflicts among some of the updates from different workers, *e.g.*, updates from different workers to the same model parameters may overwrite each other. This means that the behavior of the algorithm no

Algorithm 1 Processing an iteration as a transaction

```
1: procedure PROCESS TRANSACTION $T_i$
2:     $\mu \leftarrow P.read(T_i.read\text{-}set)$
3:     $\delta \leftarrow ML\_computation(\mu, T_i.sample, T_i.write\text{-}set)$
4:     $P \leftarrow \delta$
```

Algorithm 2 Parallel machine learning algorithm with Optimistic Concurrency Control

```
procedure PROCESS TRANSACTION $T_i$
    $\mu \leftarrow P.read_{versioned}(T_i.read\text{-}set)$
    $\delta \leftarrow ML\_computation(\mu, T_i.sample, T_i.write\text{-}set)$
    ATOMIC{
        $P.validate(\mu.versions)$
        if not validated: abort or restart
        $P \leftarrow \delta$
    }
```

longer resembles the intended serial execution of the machine learning algorithm.

Figure 3(a) illustrates the possibility of data corruption. Three workers are depicted processing three iterations of a machine learning algorithm concurrently. Each iteration reads a subset of the model parameters, computes new estimates of a subset of the model parameters, and finally writes them. In the figure, iterations 1 and 3 read and update the model parameter p and iteration 2 reads and updates the model parameter q. Iterations 1 and 3 read the same version of the parameter p, denoted p_0, and use it to calculate the new parameter value of p. Worker 1 writes the new state of p denoted p_1 and then worker 3 writes the new state of p denoted p_3. In this scenario, the work of worker 1 is overwritten by worker 3. Meanwhile, iteration 2 reads and updates parameter q, which does not corrupt the work of other iterations because it is not reading or writing parameter p.

Serializability can be the correctness criterion for parallel machine learning algorithms [21]. Serializability theory abstracts access to shared data by using the concept of a transaction [10] where a transaction is a collection of read and write operations on data objects. A data object is a subset of the shared state. The computation of an iteration i of a machine learning algorithm may be abstracted as a transaction, T_i, by considering reads of the model parameters as reads of data objects and writes to the model parameters as writes to data objects. We will denote the collection of model parameters by P, where $P[x]$ is the value of model parameter x. The parameters that are read by a transaction are denoted as $T_i.read\text{-}set$. Similarly, we will denote the parameters that are written by the transaction as $T_i.write\text{-}set$. The sample's data that is processed by iteration i is denoted by $T_i.sample$, where i is the id of the transaction. In the rest of the paper, we will use the terminology of transactions when appropriate, where a transaction is an iteration, and a data object is a model parameter.

The processing of a transaction follows the template in Algorithm 1 which is a transaction processing template that does not perform any coordination and is only serializable if run sequentially. The transaction template algorithm first reads the model parameters declared in the read-set and cache them locally as μ (line 2). Then, the read parameter values μ and the data sample information, $T_i.sample$, are used to compute new values of the parameters declared in the write-set (line 3). The new values are computed according to the used machine learning algorithms, and they are buffered locally as δ. Finally, the new parameter values, δ, are applied to the shared model parameters, P (line 4).

Serializability guarantees the illusion of a serial execution while being oblivious of the semantic computation performed within the transaction. Thus, it may be applied to machine learning algorithms. Serializability is achieved by ensuring that if some transactions conflict with each other, then they will not be executed concurrently. Detecting conflicts between concurrent transactions requires coordination among workers via different methods. These methods are diverse with different performance characteristics. We now present common transaction execution protocols that have been used in the context of machine learning algorithms, and we generalize them as transactional patterns that are oblivious to the machine learning algorithm. Readers familiar with transaction processing may skip to Section 2.3.

2.2.1 *Locking*

One of the most common methods for transaction management is *mutual exclusion* also known as lock-based protocols or pessimistic concurrency control [8]. In the rest of the paper, we will call it *Locking*. Locking is used in many parallel machine learning frameworks to support serializability [9, 19]. In this method, all read or written model parameters are locked during the processing of the transaction. These locks prevent any two transactions from executing concurrently if they access any common objects. Locking may be applied to the transactional pattern of Algorithm 1 by locking all data objects in the read-set and write-set at the beginning. These locks are released only after the transaction updates are applied to the shared model parameters.

Locking prevents conflicts such as the overwrite of worker 3 to worker 1's work in the scenario in Figure 3(a). The scenario with Locking is shown in Figure 3(b). Workers attempt to acquire a lock on the parameters they read or write before beginning the iteration. Worker 1 acquires the lock for p first and proceeds to compute and update the value of p before releasing the lock. Thus, it prevents worker 3 from overwriting its work because worker 3 will wait until it acquires the lock. Meanwhile, worker 2 acquires the lock for q and process iteration 2 because no other iteration is reading or updating q. This is a serializable execution because it resembles the serial execution of iteration 1, iteration 2, and then iteration 3. However, locking is an expensive operation that leads to a significant performance overhead even for iterations that do not need coordination, such as iteration 2.

2.2.2 *Optimistic Concurrency Control*

Optimistic concurrency control (OCC) [15] is an alternative to Locking. It performs better for scenarios with low contention, which made it more suitable for machine learning algorithms [21]. However, existing OCC methods for machine learning applications have been only proposed as specialized algorithms for domain-specific machine learning problems. Pan et. al. [21], for example, propose optimistic concurrency control patterns for DP-Means, BP-Means, and online facility location. Unlike these specialized OCC algorithms we present a generalized OCC pattern that can be applied to arbitrary machine learning algorithms.

A general OCC protocol [15] proceeds in three phases shown in Algorithm 2 :

- *Phase I (Execution):* in the execution phase, the transaction's read-set is read from the shared model parameters (line 2). Model parameters in OCC are versioned, where the version number of a parameter is the id of the transaction that wrote it. The read parameter values and the sample information are then used in the machine learning computation (denoted ML_computation) to generate the updates to model parameters, δ (line 3). Note that during this phase no coordination or synchronization is performed.
- *Phase II (Validation):* in this phase we ensure that the read data objects were not overwritten by other transactions during the execution phase (lines 5-6). This is performed by reading the model parameters again after the computation and comparing the read versions to the current versions.
- *Phase III (Commit):* if the validation is successful, the updates, δ, are applied to the global model parameters (line 7).

One requirement for OCC to be serializable is to *perform the validation and commit phases atomically* (lines 4-8) [3]. To perform these two steps atomically, there are two typical approaches: (1) Execute these steps serially at a coordinator node. This, however, limits scalability, because it means that there is a dedicated worker that is doing the validation and commit for all iterations. Such a method can only be made efficient with domain knowledge about the machine learning problem, which means that the algorithm no longer becomes a general OCC scheme but rather a specialized OCC algorithm [21]. (2) The general approach for validation is to lock the write-set. This is different from Locking in two ways: locks are only held *after* the computation has been performed and only the data objects in the write-set are locked (data objects in the read-set are not locked). Thus, OCC outperforms Locking for cases when the contention is lower, and the write-set is significantly smaller than the read-set. This approach is adopted by recent state-of-the-art OCC transaction protocols in the systems and database systems community [29] and is the method we use in our evaluations.

2.3 Performance and Overheads of Consistency Schemes

Consistency schemes, such as Locking and OCC, incur overheads to ensure a serializable execution. These overheads are: (1) *Conflict detection overhead*: this is the overhead due to additional operations needed to detect conflicts, such as locks, atomic sections, and comparing versions. These overheads are incurred even in the absence of a conflict. (2) *Backoff overhead*: this is the wasted time that is incurred due to a detected conflict, such as waiting for a lock to be released, aborting due to deadlock, and failed validation.

For Locking, the conflict detection overhead is due to the operations to acquire and release locks. Even in the absence of conflict, these operations incur an overhead. The backoff overhead for Locking is the time spent waiting for acquired locks to be released. Deadlocks do not occur in our Locking algorithm. This is because locks are acquired in ascending order—locks with lower keys are acquired first. This is possible because the read and written data objects are declared at the beginning of the execution.

For OCC, the conflict detection overhead is due to the operations to acquire and release locks for the atomic section, and the overhead to validate the read-set. Unlike Locking, OCC locks are only for the data objects in the write-set. The backoff overhead for OCC is due to wasted processing time in the case of an abort and restart when validation fails.

3. CONFLICT ORDER PLANNING

In this section, we propose Conflict Order Planning (COP) for parallel machine learning that ensures a serializable execution while reducing the overhead of conflict detection. COP entails no use of locks or atomic blocks, which are expensive operations necessary for existing consistency schemes such as Locking and OCC.

3.1 Overview

COP leverages the dataset knowledge property of machine learning workloads: a machine learning algorithm processes a dataset of samples that is known prior to the experiment and is typically processed *multiple* times. This creates the opportunity to plan a partial order of execution to minimize the cost of conflict detection. Dataset knowledge is not manifested in traditional database systems. Thus, existing consistency schemes, such as Locking and OCC are designed with the assumption that they are oblivious of the dataset. The use of traditional database transactional methods leads to a lost opportunity as they do not exploit dataset knowledge. In this section, we propose COP algorithms that exploit dataset knowledge.

The intuition behind COP is to have a planned partial order of transactions prior to execution and then ensure that the partial order is followed during execution. We derive the planned partial order from an arbitrary starting serial order of transactions. For example, a dataset with n samples will be transformed to n transactions in some planned order $T_1, T_2, \dots T_n$. We will represent this ordering by the relation $T_i <_o T_j$, where T_i is ordered before T_j. However, during execution, *the order is not enforced between every pair of transactions.* Rather, the order is only enforced for transactions that depend on each other. Thus, if T_2 does not depend on T_1 then a worker may start processing T_2 even if T_1 did not finish. Otherwise, processing T_2 must begin only after T_1 finishes. Thus, the enforced partial order is based on an initial serial order and the conflict relations between transactions.

Definition 1. ***(Planned partial order)*** *There is a planned dependency—or dependency for short—from a transaction T_i to a transaction T_j if the planned order entails T_j reading or overwriting a write made by T_i. We denote this dependency by $T_i \rightsquigarrow_x T_j$ and it exists if all the following conditions are met:*

- *T_i writes the model parameter x ($x \in T_i.write\text{-}set$).*
- *T_j reads or writes the model parameter x ($x \in T_j.read\text{-}set \cup T_j.write\text{-}set$).*
- *T_i is ordered before T_j ($T_i <_o T_j$).*
- *There exists no transaction T_k that is both ordered between T_i and T_j and writes x ($\nexists T_k | x \in T_k.write\text{-}set \wedge T_i <_o T_k <_o T_j$).*

Enforcing the order between transactions that depend on each other is a sufficient condition to guarantee a serializable execution (see Section 4.1 for a correctness proof).

Algorithm 3 The COP partial order planning algorithm that is performed prior to the experiment.

```
1: $Planned\_version\_list$ := A list to assign read and write versions initially all zeros
2: $version\_readers$ := A list to count the number of transactions that read a version
3: for $T_i \in$ $Dataset\ transactions$ do
4:     for $r \in T_i.read\text{-}set$ do
5:         $r.planned\_version = Planned\_version\_list[r.param]$
6:         $version\_readers[r.param]{+}{+}$
7:     for $w \in T_i.write\text{-}set$ do
8:         $w.p\_writer = Planned\_version\_list[w.param]$
9:         $Planned\_version\_list[w.param] = i$
10:        $w.p\_readers = version\_readers[w.param]$
11:        $version\_readers[w.param] = 0$
12: Delete $Planned\_version\_list$ and $version\_readers$
```

COP enforces dependencies by versioning model parameters with the ids of the transactions that wrote them. A transaction only starts execution if the versions it depends on has been written. Consider applying COP to the scenario in Figure 3(a). The resulting execution is shown in Figure 3(c). Assume that the planned order is to execute samples 1, 2, and 3, in this order. The partial order consists of a single dependency from iteration 1 to iteration 3, because they both read and write p. Iterations use a special read operation called ReadWait that waits until the version it reads is written by the transaction that it depends on. Iteration 1 is planned to read the initial version of p, denoted p_0, because it is the first ordered iteration to read p. Likewise, iteration 2 is planned to read the initial version of q. Iteration 3 depends on Iteration 1, because they both read and write p. Thus, iteration 3 is planned to read the version of p that is written by iteration 1, denoted p_1. With this plan, workers 1 and 2 process iterations 1 and 2 concurrently after verifying that they have read their planned versions. Worker 3, however, waits until the version p_1 is written by worker 1 and then proceeds to process iteration 3. With COP, workers coordinate without the need of expensive locking primitives. Rather, workers only utilize simple arithmetic operations on the read or written parameter's version number to enforce the plan.

In the remainder of this section, we propose the COP planning algorithm that is used to find and annotate dependency relations between transactions (Section 3.2). Then we propose the COP transaction execution algorithm that enforces dependency relations (Section 3.3). We discuss the performance benefits of COP in Section 3.4.

3.2 COP Planning Algorithm

In this section, we present the COP planning algorithm in its basic form—planning prior to execution. Then, we discuss how it can be used to plan in conjunction with the first epoch and how it can be used in cases where there are multiple sources of data.

3.2.1 Basic COP Planning

We begin by presenting the basic COP planning strategy. Here, we assume that planning is performed before execution, either in offline settings or while loading the dataset. The objective of the planning algorithm is to annotate the dataset with the planned partial order information. This annotation includes the following:

Definition 2. ***(COP planning and annotation)*** *COP planning performs the following two annotations: (1) Read annotation: each read operation is annotated with the version number it should read, and (2) Write annotation: each write operation, w, is annotated with the id of the version it should overwrites, w', and the number of transactions that are planned to read the version w'.*

The read annotation's goal is to enforce the order during execution. The write annotation's goal is to ensure that a version is not overwritten until it is read from all the transactions that are planned to read it.

Algorithm 3 shows the steps to annotate transactions with the partial order information. The algorithm processes transactions one transaction at a time ordered by some arbitrary order—beginning with T_1 and ending with T_n.

In COP, each read operation in the read-set, r, contains both the read parameter to be read ($r.param$), and the planned read version number ($r.planned\_version$), *i.e.*, the read annotation. A planned version number k means that the transaction must read the value written by transaction T_k. Also, each write in the write-set, w, contains the parameter to be written ($w.param$), the number of transactions that read the previous version ($w.p\_readers$), and the transaction id of the transaction that it is overwriting ($w.p\_writer$), *i.e.*, the write annotation.

The planning algorithm tracks the planned version numbers in a list named $Planned\_version\_list$ as dependencies are being processed. $Planned\_version\_list[x]$ contains the unique transaction id of the most recently planned transaction that writes x. All entries in the list are initialized to 0. Also, the number of version readers are maintained in a list named $version\_readers$. At any point in the planning process, $version\_readers[x]$ contains the number of planned transactions that read the most recently planned written version of x. Both lists are only used within the planning algorithm and are deleted before the execution phase.

The planning of a transaction T_i proceeds by processing the read-set and then the write-set. Each read operation r in the read-set is annotated with a planned version from the $Planned\_version\_list$ (lines 4-5). For example, consider the case where T_i reads model parameter x. Then, there is a read, r, with $r.param$ equals to x. At the time r is being planned, the corresponding value in the list, $Planned\_version\_list[x]$ contains the unique transaction id, k, of the last transaction, T_k, that wrote x. Thus, assigning the planned version of r to k is a way of encoding that the plan is for T_i to read the value of x that was written by T_k. Then, the corresponding number of version readers is incremented (line 6). After processing the read-set, the planning algorithm processes each write w in the write-set (lines 7-11). Each write is annotated with the previous writer's version number (line 8). Then, the corresponding entry in $Planned\_version\_list$ is updated with the transaction id value i (line 9). Thus, reads of transactions ordered after T_i can observe that they are planned to read T_i's writes. Then, the write is annotated with the number of readers of the previous version (line 10). Finally, the corresponding entry in $version\_readers$ is reset. After all the operations are processed, the lists $Planned\_version\_list$ and $version\_readers$ are deleted.

The outcome of the algorithm is read and write annotations

Algorithm 4 Parallel execution with COP

```
1: Global $num\_reads$ := initially all zeros
2: procedure PROCESS TRANSACTION $T_i$
3:     for $r \in T_i.read\text{-}set$ do
4:         $\mu \leftarrow P.ReadWait(r)$
5:         $num\_reads[r.param]{+}{+}$
6:     $\delta \leftarrow$ ML_computation ($\mu$, $T_i.sample, T_i.write\text{-}set$)
7:     for $w \in \delta$ do
8:         $w.version = i$
9:         while $w.p\_readers \neq num\_reads[w.param]$ OR $w.p\_writer \neq P[w.param].version$ do
10:            Wait
11:        $num\_reads[w.param] = 0$
12:    $P \leftarrow \delta$
```

of the whole dataset. The algorithm only requires a single pass on the dataset. In the evaluation section, we perform experiments to quantify the overhead of planning.

3.2.2 Alternative Planning Strategies

The basic COP planning algorithm, presented in the previous section, assumes that planning is performed prior to execution in offline settings or during dataset loading. We now show how to adapt the algorithm to plan in alternative planning scenarios. The first alternative is to plan during the first epoch of the machine learning algorithm's execution. The plan's objective is to annotate transactions with a partial order of a serializable execution. It is possible to execute the first epoch of the machine learning algorithm via a traditional consistency scheme (*e.g.*, Locking) and then annotate the dataset with the partial order of that epoch. Specifically, during the first epoch using Locking, the planning Algorithm 3 is performed for each transaction while all the locks of that transaction are held. Thus, each read is annotated with the read version and each write is annotated with the version it overwrites and the number of readers. After the first epoch, that has passed through the whole dataset, the remaining epochs are processed using COP with the annotated plan. The planning only adds a small overhead to the first epoch, as we discuss in the evaluation section.

Another alternative is to plan when the dataset is being generated online from multiple sources, in cases such as the global analytics scenario in Section 2.1.2. In such a scenario, planning can be done at each source for batches of samples using Algorithm 3. Then, at the centralized location, the machines learning algorithms process batches in tandem. The dependencies of a batch are transposed to previous batches. For example, consider two batches b_1 and b_2, where b_1 is processed in the centralized location prior to b_2. The transactions in b_2 that have dependencies on the initial version, according to Algorithm 3, are transposed to the most recent version written by b_1. For example, the first transaction that accesses x in b_2 will be annotated as reading the version 0. However, the centralized location will translate this as an annotation to wait for the last version written by b_1.

3.3 Planned Execution Algorithm

We present the COP execution algorithm (shown in Algorithm 4) that processes transactions in parallel according to a planned partial order. We associate each model parameter with a version number that corresponds to the transaction that wrote it, *e.g.*, $P[x].version$ is the current version number of model parameter x. A list of the number of version readers for model parameters, num_reads, is maintained and accessible by all workers. For example, a value for $num\_reads[x]$ of 3 means that so far, three transactions read the current version of x.

Dependencies between transactions are enforced by ensuring that read operations read the planned versions. T_i's read-set is read from the shared model parameters, P (lines 3-5). The ReadWait operation blocks until the annotated planned version is available. The implementation of ReadWait simply reads both the data object and its version number. Then, it compares the version number to the annotated read version number. If they match, the read data object is returned; otherwise, the read is retried until the planned version is read.

After reading the planned version, the number of version readers is incremented (line 5). Then, the transaction execution proceeds by performing the machine learning computation using the read model parameters and the data sample's information (line 6). Writes to the model parameters computed by the machine learning computation, δ, are buffered before they are applied to the model parameters (lines 7-11). First, each write, w, is tagged with a version number equal to the transaction's id (line 8). Thus, future transactions that read the state can infer that T_i is the transaction that wrote these updates. Then, the algorithm waits until the previous version has been read by all planned readers by making sure that the number of version readers is equal to the planned number of readers of that version and by making sure that the current version is identical to $w.p\_writer$ (lines 9-10). Since we are writing a new version, the corresponding entry in num_reads is reset to 0. The writes are then incorporated in the shared state (line 12).

3.4 Performance and Overheads

In Section 2.3 we discussed two overheads of consistency schemes: conflict detection overhead and backoff overhead. The backoff overhead incurred in COP is similar to Locking and OCC, *i.e.*, transactions wait until conflicting transactions complete. COP's goal is to minimize the other source of overhead: conflict detection overhead that is incurred whether a conflict is detected or not. In COP, the conflict detection overhead is due to: (1) The validation using the ReadWait operation, and (2) Validation that each write operation's previous readers have already read the previous version. These two tasks are performed via arithmetic operations and comparisons only, without the need for expensive synchronization operations like acquiring and releasing locks. This is the main contributor to COP's performance advantage.

4. CORRECTNESS PROOFS

In this section, we present two proofs. The first proves that COP is serializable and the second proves that deadlocks do not occur in COP.

4.1 COP Serializability

We prove the correctness of COP and that it ensures a serializable execution that is equivalent to a serial execution. We use a serializability graph (SG) to prove COP's serializability [3]. A protocol is proven serializable if the SGs that represent its possible executions do not have cycles.

A SG consists of nodes and edges. Each node represents a committed transaction. A directed edge from one node to another represents a conflict relation. There are three types of conflict relations (edges) in SGs:

- *Write-read (wr) relation*: This relation is denoted as $T_i \rightarrow_{wr} T_j$, which means that there is a *wr* edge from T_i to T_j. This relation exists if T_i writes a version of a data object x and T_j reads that version.
- *Write-write (ww) relation*: This relation is denoted as $T_i \rightarrow_{ww} T_j$, which means that there is a *ww* edge from T_i to T_j. This relation exists if T_i writes a version of a data object x and T_j overwrites that version with a new one.
- *Read-write (rw) relation*: This relation is denoted as $T_i \rightarrow_{rw} T_j$, which means that there is a *rw* edge from T_i to T_j. This relation exists if T_i reads a version of a data object x and T_j overwrites that version with a new one. If this edge exists between two transactions ($T_i \rightarrow_{rw} T_j$) then it must be the case that there exists a transaction T_k that writes x with the following conflict relations: (1) A write-write conflict relation from T_k to T_j ($T_k \rightarrow_{ww} T_j$), and (2) a write-read conflict relation from T_k to T_i ($T_k \rightarrow_{wr} T_i$).

Lemma 1. *For any conflict relation $T_i \rightarrow T_j$ in SG of a COP execution, the following is true: $T_i <_o T_j$, where $<_o$ is the ordering relation of the initial planned order.*

Proof. Assume that the data object that causes the conflict relation is data object x. We prove this lemma for the three conflict relations:

- *Write-read (wr) conflict relations ($T_i \rightarrow_{wr} T_j$):* according to Definition 1 a transaction T_j is planned to read from a transaction T_i if there is an ordering dependency $T_i \rightsquigarrow_x T_j$. One of the conditions of this ordering dependency is that T_i is ordered before T_j ($T_i <_o T_j$). In the implementation algorithm, this is enforced by the ReadWait operation (see Algorithm 4 lines 3-4).
- *Write-write (ww) conflict relations ($T_i \rightarrow_{ww} T_j$)):* according to Definition 1 a transaction T_j is planned to overwrite a value written by transaction T_i if there is an ordering dependency $T_i \rightsquigarrow_x T_j$. One of the conditions of this ordering dependency is that T_i is ordered before T_j ($T_i <_o T_j$). In the implementation algorithm, this is enforced by the check of *w.p_writer* (see Algorithm 4 lines 9-10).
- *Read-write (rw) conflict relations ($T_i \rightarrow_{rw} T_j$):* this relation implies the existence of a transaction T_k with the relations $T_k \rightarrow_{wr} T_i$ and $T_k \rightarrow_{ww} T_j$. According to our analysis in the previous two points, the following is true:

  $$T_k <_o T_i \quad and \quad T_k <_o T_j \tag{1}$$

  Thus, the following ordering dependencies exist:

  $$T_k \rightsquigarrow_x T_i \quad and \quad T_k \rightsquigarrow_x T_j \tag{2}$$

  We now show by contradiction that the following is true: $T_i <_o T_j$. Assume to the contrary that $T_j <_o T_i$ is true. If $T_j <_o T_i$ then according to Equation 1 the following is true:

  $$T_k <_o T_j <_o T_i \tag{3}$$

  However, this equation violates one of the definitions in Definition 1 that states that the ordering relation $T_k \rightsquigarrow_x T_i$ that exists according to Equation 2 implies that there exists no transaction that is ordered between them and writes x. However, according to Equation 3, T_j is ordered between T_k and T_i and it writes x. This violation leads to a contradiction to $T_j <_o T_i$ thus proving that $T_i <_o T_j$. In the implementation algorithm, this is enforced by the check of *w.p_readers* (see Algorithm 4 lines 9-10).

The condition of the lemma is proven for all three conflict relations. □

Theorem 1. *Conflict Order Planning (COP) algorithms guarantee serializability.*

Proof. According to Lemma 1, a conflict relation $T_i \rightarrow T_j$ in SG means that $T_i <_o T_j$. We need to show that a cycle $T_i \rightarrow \ldots \rightarrow T_i$ cannot exist. Assume to the contrary that such a cycle exists. This means according to Lemma 1 that $T_i <_o \ldots <_o T_i$. Since the ordering relation $<_o$ is transitive this leads to $T_i <_o T_i$, which is a contradiction, thus proving that no cycles exist in the SG of COP executions. The absence of cycles in SG is a sufficient condition to prove serializability [3]. □

4.2 COP Deadlock Freedom

The COP execution algorithm 4 can block in two locations: (1) a read waits for its planned version to be available, and (2) a write waits until all reads of the previous versions and the write of the previous version are complete. In this section we prove that these waits do not cause a deadlock scenario where a group of transactions are waiting for each other. We prove this by constructing a deadlock graph (DG). Nodes in DG are transactions. A directed edge from one transaction to another, $T_i \rightarrow_d T_j$, denotes that T_j may block waiting for a read or a write of T_i.

Lemma 2. *For any edge in DG, $T_i \rightarrow_d T_j$, the following is true: $T_i <_o T_j$, where $<_o$ is the ordering relation of the planned order.*

Proof. An edge $T_i \rightarrow_d T_j$ exists in three cases: (1) T_j reads a version written by T_i, (2) T_j overwrites a version written by T_i, or (3) T_j overwrites a version to be read by T_i. All cases are true in the COP algorithms only if the ordering dependency $T_i \rightsquigarrow T_j$ exists. According to Definition 1, an ordering dependency $T_i \rightsquigarrow T_j$ is only true if $T_i <_o T_j$. □

Theorem 2. *Conflict Order Planning (COP) algorithms guarantee deadlock freedom.*

Proof. According to Lemma 2, a dependency relation $T_i \rightarrow_d T_j$ in DG means that $T_i <_o T_j$. We need to show that a cycle $T_i \rightarrow_d \ldots \rightarrow_d T_i$ cannot exist. Assume to the contrary that such a cycle exists. This means according to Lemma 2 that $T_i <_o \ldots <_o T_i$. Since the ordering relation $<_o$ is transitive this leads to $T_i <_o T_i$, which is a contradiction, thus proving that no cycles exist in the DG of COP executions. The absence of cycles in DG is a sufficient condition to prove deadlock freedom. □

	Properties				Performance (M transactions/s)			
Dataset	# features	training set size	test set size	avg. transaction size	Ideal	COP	Locking	OCC
KDDA [26]	20,216,830	8,407,752	510,302	36.3	7.2	4.1	0.75	0.82
KDDB [26]	29,890,095	19,264,097	748,401	29.4	8.0	5.8	0.95	1.0
IMDB	685,569	167,773		14.6	15.2	11.0	6.7	4.9

Table 1: Performance comparison across of COP, Locking, OCC, and Ideal (without conflict detection) for three datasets

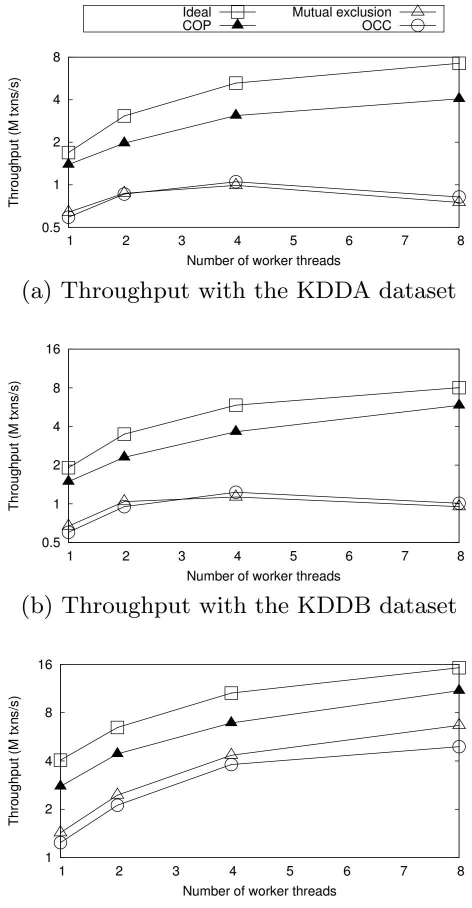
5. EVALUATION

In this section, we evaluate COP in comparison to Locking and OCC. We also compare with an upper-bound of performance, which is the performance without any conflict detection. We will call this upper-bound the *ideal* baseline, or Ideal for short. Ideal does not guarantee a serializable execution, unlike COP, Locking, and OCC. Thus, Ideal does not guarantee preserving the theoretical properties of the machine learning algorithm.

The transactional framework of machine learning can be applied to a wide-range of machine learning algorithms. For this evaluation, we run our experiments with a Stochastic Gradient Descent (SGD) algorithm to learn a Support Vector Machine (SVM) model. The goal of the machine learning algorithm is to minimize a cost function f. We use a separable cost function for SVM [25]. Each iteration in SGD processes a single sample from the dataset. Gradients are computed according to the cost function. The gradients are then used to compute the new values of the model that are relevant to the sample. We apply this machine learning algorithm to the transactional template we presented in Algorithm 1. Each transaction corresponds to an iteration of SGD. The iteration computation (*i.e.*, ML_computation() in the algorithm) represents the gradient computation using the cost function. For this machine learning algorithm, the read and write-sets of a transaction are the features in the corresponding sample, *i.e.*, the features with a non-zero value. In all experiments, we initialize the SGD step size value to 0.1. The step size value diminishes by a factor 0.9 at the end of each epoch over the training dataset. All experiments are run for 20 epochs, where an epoch is a complete pass on the whole dataset.

We implemented COP, Locking, and OCC as a layer on top of the parallel machine learning framework of Hogwild! [25] that is available publicly[1]. The source code is written in C++. We use an Amazon AWS EC2 virtual machine to run our experiments. The virtual machine type is c4.4xlarge with 30 GB memory and 16 vCPUs that are equivalent to 8 physical cores (Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz) and 16 hyper threads. Unless mentioned otherwise, the number of worker threads used in the experiments is 8. Our experiments with more than 8 threads show no significant performance difference.

We use three datasets to conduct our experiments, summarized in Table 1. The first two datasets are KDDA and KDDB datasets, which were part of the 2010 KDD Cup [26]. KDDA (labeled algebra_2008_2009) has 20,216,830 features and contains 8,407,752 samples in the training set and 510,302 samples in the test set. KDDB (labeled bridge_to_algebra_2008_2009) has 29,890,095 features and contains 19,264,097 samples in the training set and 748,401 samples in the test set. The third dataset is the IMDB dataset[2] that has 685,569 features and contains 167,773 samples. The IMDB dataset is not divided into training and test sets. The average sample (transaction) size of each dataset, which is the number of model parameters represented in each sample, is 36.3 for KDDA, 29.4 for KDDB, and 14.6 for IMDB. In addition to these three datasets, we use synthetic datasets for experiments that require controlling the dataset properties, such as contention.



(a) Throughput with the KDDA dataset

(b) Throughput with the KDDB dataset

(c) Throughput with the IMDB dataset

Figure 4: The throughput of Ideal, COP, Locking, and OCC while varying the number of threads for three datasets (log scale is used)

5.1 Throughput

The metric that we are interested in the most is throughput. We measure throughput as the number of processed samples

[1] http://i.stanford.edu/hazy/victor/Hogwild/

[2] http://komarix.org/ac/ds/

(*i.e.*, transactions) per second. Table 1 shows a summary of throughput numbers for the different evaluated methods on the three datasets. COP outperforms Locking and OCC by a factor of 5-6x for KDDA and KDDB. For IMDB, COP's throughput is 64% higher than Locking and 124% higher than OCC. The magnitude of performance improvement of COP compared to Locking and OCC is influenced by the level of contention in the dataset, *i.e.*, the likelihood of conflict between transactions. Our inspection of the datasets revealed that there is more opportunity for conflict in the KDDA and KDDB datasets than the IMDB dataset. We do not present the statistical properties of the datasets to show this due to the lack of space. However, we perform more experiments in Section 5.2 to study the effect of contention on performance. The comparison with Ideal shows that COP's throughput is 27-44% lower than Ideal. This percentage represents COP's overhead to preserve consistency. Although conflicts are planned in COP, there is still an overhead for conflict detection and backoff.

The throughputs of Locking and OCC are relatively close to each other. For KDDA and KDDB, the throughputs of Locking and OCC are within 10% of each other. For IMDB, Locking outperforms OCC by 36.7%. In the case of KDDA and KDDB, the locking contention for both Locking and OCC (to implement atomic validation) dominates performance. In general, OCC benefits in cases where the read-set is larger than the write-set. Because our machine learning workload has a read and write-sets of equal sizes, the advantage of OCC is not manifested (see Section 2.3). In IMDB, which is the workload with less contention, Locking outperforms OCC. This is due to the additional work needed to validate transactions by OCC. For the conflict detection overhead, OCC experiences both the overheads of locking and validation, while Locking only experiences the overhead of locking. The overhead of validation is exposed with workloads with less contention because in these cases, locking contention does not dominate performance, *i.e.*, in the case of the KDDA and KDDB datasets, the overhead due to locking contention dominates the validation overhead. We revisit the effect of contention in Section 5.2.

In Figure 4, we show the performance of the different schemes while varying the number of threads. Increasing the number of threads increases contention. Also, using more cores in the experiment exposes the effect of the underlying cache and cache coherence on the performance of the different schemes. Figure 4(a) shows the performance for the KDDA dataset. Consider the throughput of all schemes with a single worker thread. In this case, there is no conflict or cache coherence overhead. What is observed is the *conflict detection* overhead in isolation (Section 2.3). Ideal is only 21% higher than COP in the case of a single worker thread. This shows that the overhead of conflict detection is small compared to Locking and OCC; the throughput of Ideal is 163% higher than Locking and 186% higher than OCC.

For scenarios with more than one worker thread in Figure 4(a), the backoff and cache coherence overheads are experienced in addition to the conflict detection overhead. Ideal does not suffer from the backoff overhead because conflicts are not prevented. Also, Ideal has an advantage with the cache coherence overhead compared to the consistent schemes. Unlike COP, Locking, and OCC, Ideal does not maintain additional locking or versioning data that may be invalidated by cache coherence protocols. These factors cause the performance gap between Ideal and the other schemes to grow as the number of threads is increased. COP's throughput, for example, is 17% lower than Ideal with one worker thread, but it is lower by 43% in the case of 8 threads. The contention between cores due to cache coherence limits scalability. Ideal with 8 threads achieves 4 times the performance of the case with a single thread—rather than 8 times the performance in the case of linear scalability. COP with 8 threads achieves 3 times the performance of the case with a single thread. For Locking and OCC, the contention is so severe that performance slightly decreases beyond 4 threads.

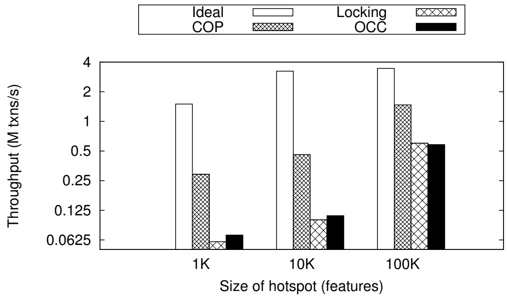


Figure 5: Quantifying the effect of contention on performance by experiments on synthetic datasets with varying contention levels

We show the same set of experiments for KDDB and IMDB in Figures 4(b) and 4(c). The experiments with the KDDB dataset show similar behavior to the experiments with the KDDA dataset. One difference is that COP scales better, as the KDDB dataset is sparser than KDDA; for KDDB, COP's throughput with 8 threads is 4 times the throughput with a single thread, rather than a 3x factor with the KDDA dataset. For the IMDB dataset, there is less contention compared to KDDA and KDDB. All schemes—including Locking and OCC—scale with a factor around 4x when increasing the number of threads from 1 to 8. Also, the smaller transaction sizes with the IMDB dataset makes the absolute throughput numbers higher than those with the KDDA and KDDB datasets.

5.2 Contention Effect

Contention affects performance because it increases the rate of conflict. A conflict between two transactions causes at least one of them to either wait or restart, thus wasting resources. Here, we quantify the effect of contention on the performance of our consistency schemes. We generate synthetic datasets to give us more flexibility in controlling the contention. The synthetic datasets we generate contain one million samples each. We fix the size of each sample to 100 features, which means that each transaction contains 100 data objects in the read and write-sets. To control the contention, we restrict transactions to a hot spot in the parameter space. Each data object is sampled uniformly from the hot spot. We control contention by varying the size of the hot spot.

Figure 5 shows the performance with hot spot sizes of 1K, 10K, and 100K features. Contention leads to a higher conflict overhead and lower performance. This is why consistency schemes perform lower in the highest contention case (1K features) when compared to cases with less contention. The performance improvement factor of the case with 100K fea-

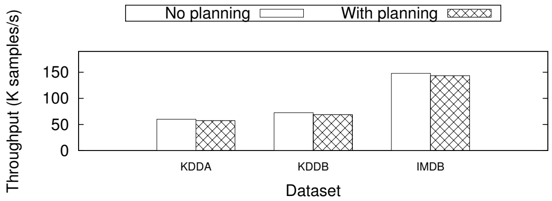


Figure 6: A comparison of the loading time of the dataset to main memory with and without order planning.

tures compared to the case with 1K features is 4x for COP, 8.8x for Locking, and 7.3x for OCC. Ideal also performs lower as contention increases, although it does not face an overhead due to conflicts. The performance of Ideal with 100K features is 131% higher than the performance with 1K features. The reason is that more contention also means more contention on cache lines, leading to a larger overhead for cache coherence.

As contention decreases, the performance gap between the consistency schemes and Ideal decreases. Part of the performance benefit of Ideal compared to the consistency schemes is that Ideal does not block or restart transactions due to conflicts. As contention decreases, this performance benefit diminishes, and the performance of the consistency schemes becomes closer to Ideal. For example, in the high contention case (1K features) Ideal's throughput is 4x the throughput of COP. For the low contention case (100K features), this gap decreases with Ideal's throughput only 34% higher than COP. This is also true for Locking and OCC, where Ideal's throughput is higher than them by a factor of 20-23x in the high contention case, but this factor decreases to around 5x for the low contention case.

Like the performance difference between Ideal and the other consistency schemes, the performance gap between COP and the other consistency schemes (Locking and OCC) also decreases as contention decreases. COP's light-weight conflict detection makes it less prone to conflicts than Locking and OCC because the latency of the transaction is lower. Thus, Locking and OCC suffer from contention more than COP. In the low contention case, COP's throughput is 3.7x higher than Locking and 3.1x higher than OCC. This performance gap decreases in the low contention case where COP outperforms Locking by 46% and OCC by 51%.

5.3 Planning Overhead

COP's performance advantage is due to having conflicts planned ahead of time. We have outlined in Section 2.1 examples of machine learning environments. In these environments, planning can be done in advance, and thus the planning overhead is not observed when the machine learning algorithm is processed using COP. This includes the case of the machine learning framework where a dataset is reused in different experiments and is possibly stored with the annotated plan for future sessions. However, there are cases where machine learning algorithms are used for fresh and raw datasets. In these cases, the planning overhead becomes important.

We performed several experiments to quantify the overhead of planning. We propose two alternatives to plan for a dataset. The first planning strategy is to plan while loading the dataset. A dataset is typically stored in a persistent storage such as a disk. Planning can be done in conjunction with reading the raw dataset from persistent storage and loading it into the appropriate data structures in main memory. Figure 6 shows the loading throughput with and without planning for three datasets. Planning only adds a small overhead to loading that we measure to be between 3% and 5%.

The second planning strategy is to plan during the first epoch and then use the plan for later epochs (Section 3.2.2). In the first epoch, a consistency scheme must be used. We run the first epoch using Locking and the rest of the epochs using COP. The throughput of the first epoch is within 1% of the throughput of Locking for all our datasets. The throughput of the remaining epoch is also within 1% of the performance of COP with offline planning.

6. RELATED WORK

The use of transactional and consistency concepts have been explored recently for parallel and distribute machine learning by Pan et.al [20–24]. Some of these works build consistent algorithms that follow the OCC pattern for distributed unsupervised learning [21], correlation clustering [20, 24], and submodular maximization [22]. These proposals show that domain-specific implementations of OCC—rather than general OCC that we presented in this work—achieve performance close to their coordination-free counterparts while guaranteeing serializability [22, 24].

The study of consistent machine learning algorithms has been motivated by the complexity of developing mathematical guarantees and coordination-free algorithms that are parallel [20–24]. However, many coordination-free machine learning algorithms were developed [1, 6, 18, 25]. Hogwild! [25], for example, is an asynchronous parallel SGD algorithm with proven convergence guarantees for several classes of machine learning algorithms.

Bounded staleness has been proposed as an alternative to both serializability and coordination-free execution for parallel machine learning. Bounded staleness is a correctness guarantee of the freshness of read data objects. It has been demonstrated for distributed machine learning tasks [13, 17]. Bounded staleness, however, may still lead to data corruption which requires a careful design of machine learning algorithms that leverage bounded staleness.

The concept of planning execution to improve the performance of distributed and parallel transaction processing has been explored in different contexts. Calvin [28] is a deterministic transaction execution protocol. Sequencing workers intercept transactions and put them in a global order that is enforced by scheduling workers. Calvin is built for typical database transactional workload and thus does not leverage the dataset knowledge property of machine learning workloads. This makes its design incur unnecessary overheads compared to COP for machine learning workloads, such as always having the sequencing and scheduling workers in the path of execution. Schism [5] is a workload-driven replication and partitioning approach. The access patterns are learned from the coming workload to create partitioning strategies that minimize conflict between partitions and thus improve performance. Cyclades [23] adopts a similar approach to Schism for parallel machine learning workloads. Cyclades improves the performance of both conflict-free and consistent machine learning algorithms by partitioning access for batches of the dataset to minimize conflict between partitions. Each partition is then processed by a dedicated thread, leading to better performance. Partitioning for performance complements COP's objective. Whereas partitioning aims to minimize conflict between workers, COP ensures that conflicts are handled more efficiently.

7. CONCLUSION

In this paper, we propose Conflict Order Planning (COP) for consistent parallel machine learning. COP leverages dataset knowledge to plan a partial order of concurrent execution. Planning enables COP to execute with light-weight synchronization operations and outperform existing consistency schemes such as Locking and OCC while maintaining serializability for machine learning workloads. Our evaluations validate the efficiency of COP on a SGD algorithm for SVMs.

8. ACKNOWLEDGMENT

This work is partially funded by a gift grant from Oracle and a gift grant from NEC Labs America. We would also like to thank Amazon for access to Amazon EC2.

References

[1] H. Avron et al. Revisiting asynchronous linear solvers: Provable convergence rate through randomization. *Journal of the ACM (JACM)*, 62(6):51, 2015.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. 1987.

[3] P. A. Bernstein and E. Newcomer. *Principles of transaction processing*. Morgan Kaufmann, 2009.

[4] I. Cano et al. Towards geo-distributed machine learning. In *Workshop on ML Systems at NIPS*, 2015.

[5] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *VLDB*, 3(1-2):48–57, 2010.

[6] C. M. De Sa, C. Zhang, K. Olukotun, and C. Ré. Taming the wild: A unified analysis of hogwild-style algorithms. In *NIPS*, pages 2674–2682, 2015.

[7] J. Dean et al. Large scale distributed deep networks. In *NIPS*, pages 1223–1231. 2012.

[8] K. P. Eswaran et al. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[9] J. E. Gonzalez et al. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[10] J. Gray et al. The transaction concept: Virtues and limitations. In *VLDB*, volume 81, pages 144–154, 1981.

[11] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[12] M. Hall et al. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[13] Q. Ho et al. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231. 2013.

[14] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, volume 1, pages 2–1, 2013.

[15] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, 1981.

[16] H. Li, A. Kadav, E. Kruus, and C. Ungureanu. Malt: distributed data-parallelism for existing ml applications. In *EuroSys*, 2015.

[17] M. Li et al. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[18] J. Liu, S. J. Wright, C. Ré, V. Bittorf, and S. Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. *Journal of Machine Learning Research*, 16(285-322):1–5, 2015.

[19] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[20] X. Pan et al. Scaling up correlation clustering through parallelism and concurrency control. *In DISCML workshop at NIPS*, 2014.

[21] X. Pan, J. E. Gonzalez, S. Jegelka, T. Broderick, and M. I. Jordan. Optimistic concurrency control for distributed unsupervised learning. In *NIPS*, pages 1403–1411. 2013.

[22] X. Pan, S. Jegelka, J. E. Gonzalez, J. K. Bradley, and M. I. Jordan. Parallel double greedy submodular maximization. In *NIPS*, pages 118–126, 2014.

[23] X. Pan, M. Lam, S. Tu, D. Papailiopoulos, C. Zhang, M. I. Jordan, K. Ramchandran, C. Re, and B. Recht. Cyclades: Conflict-free asynchronous machine learning. In *NIPS*. 2016.

[24] X. Pan, D. Papailiopoulos, S. Oymak, B. Recht, K. Ramchandran, and M. I. Jordan. Parallel correlation clustering on big graphs. In *NIPS*, pages 82–90. 2015.

[25] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701. 2011.

[26] J. Stamper, A. Niculescu-Mizil, S. Ritter, G. Gordon, and K. Koedinger. Challenge data set from kdd cup 2010 educational data mining challenge, 2010. [https://pslcdatashop.web.cmu.edu/KDDCup/].

[27] A. Talwalkar, T. Kraska, R. Griffith, J. Duchi, J. Gonzalez, D. Britz, X. Pan, V. Smith, E. Sparks, A. Wibisono, et al. Mlbase: A distributed machine learning wrapper. *Big Learning Workshop at NIPS*, 2012.

[28] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[29] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32, 2013.

ChronicleDB: A High-Performance Event Store

Marc Seidemann
Database Systems Group
University of Marburg, Germany
seidemann@informatik.uni-marburg.de

Bernhard Seeger
Database Systems Group
University of Marburg, Germany
seeger@informatik.uni-marburg.de

ABSTRACT

Reactive security monitoring, self-driving cars, the Internet of Things (IoT) and many other novel applications require systems for both writing events arriving at very high and fluctuating rates to persistent storage as well as supporting analytical ad-hoc queries. As standard database systems are not capable to deliver the required write performance, log-based systems, key-value stores and other write-optimized data stores have emerged recently. However, the drawbacks of these systems are a fair query performance and the lack of suitable instant recovery mechanisms in case of system failures.

In this paper, we present ChronicleDB, a novel database system with a well-designed storage layout to achieve high write-performance under fluctuating data rates and powerful indexing capabilities to support ad-hoc queries. In addition, ChronicleDB offers low-cost fault tolerance and instant recovery within milliseconds. Unlike previous work, ChronicleDB is designed either as a serverless library to be tightly integrated in an application or as a standalone database server. Our results of an experimental evaluation with real and synthetic data reveal that ChronicleDB clearly outperforms competing systems with respect to both write and query performance.

1. INTRODUCTION

Data objects in time also known as events are ubiquitous in today's information landscape. They arise at lower levels in the context of operating systems like file accesses, CPU usage or network packets, but also at higher levels, for example, in the context of online shopping transactions. The rapidly growing *Internet of Things (IoT)* is reinforcing the new challenge for present-day data processing. Sensor-equipped devices are becoming omnipresent in companies, smart buildings, airplanes or self-driving cars. Furthermore, scientific observational (sensor) data is crucial in various research, spanning from climate research over animal tracking to IT security monitoring. All these applications rely on low-latency processing of events attached with one or multiple temporal attributes.

Due to the rapidly increasing number of sensors not only the analysis of such events, but also their pure ingestion is becoming a big challenge. On-the-fly event stream processing is not always the outright solution. Many fields of application require to maintain the collected historical data as time series for the long term, e.g., for further temporal analysis or provenance reasons. For example, in the field of IT security, historical data is crucial to reproduce critical security incidents and to derive new security patterns. This requires writing various sensor measurements arriving at high and fluctuating speed to persistent storage. Data loss due to system failures or system overload is generally not acceptable as these data sets are of utmost importance in operational applications.

There is a current lack of systems supporting the above described workload scenario (write-intensive, ad-hoc temporal queries and fault-tolerance). Standard database systems are not designed for supporting such write-intensive workloads. Their separation of data and transaction logs generally incurs high overheads. Our experiments with traditional relational systems revealed that their insertion performance is insufficient to keep up with the targeted data rates. PostgreSQL [9], for example, managed only about 10K tuple insertions per second. Relational database systems are designed to store data with the focus on query processing and transactional safety. Instead of relational systems, today, it is common to use distributed key-value systems like Cassandra [1] or HBase [3], but they only alleviate the write problem at a very high cost. In our benchmarks, our event store *ChronicleDB* outperformed Cassandra by a factor of 47 in terms of write-performance on a single node. In other words: Cassandra would need at least 47 machines to compete with our solution. Apart from economical aspects due to high expenses for large clusters, there are many embedded systems where scalable distributed storage does not suit. For example, in [14], virtual machines of a physical server are monitored within a central monitoring virtual machine, which does not allow a distributed storage solution due to security reasons. Other examples are self-driving cars and airplanes that need to manage huge data rates within a local system.

In order to overcome these deficiencies, particularly the poor write performance, log-only systems have emerged where the log is the only data repository [32, 16, 33]. Our system ChronicleDB also keeps its data in a log, but it differs from previous approaches by its centralized design for

Series ISSN: 2367-2005

10.5441/002/edbt.2017.14

high volume event streams. Because there are often only modest changes in event streams, ChronicleDB exploits the great potential of their lossless compression to boost write and read performance beyond that of previous log-only approaches. This also requires the design of a novel storage layout to achieve fault tolerance and near-instant recovery within milliseconds in case of a system failure. In addition to lightweight temporal indexing, ChronicleDB offers adaptive indexing support to significantly speed-up non-temporal queries on its log. ChronicleDB can either be plugged into applications as a library or run as a centralized system without the necessity to use the common distributed storage stack. In summary, we make the following contributions:

- We propose an efficient and robust storage layout for compressed data with fault tolerance and instant recovery.
- ChronicleDB offers an adaptive indexing technique compromising both lightweight temporal indexing as well as full secondary indexing to speed-up queries on non-temporal dimensions.
- In order to support out-of-order arrival of events, we developed a hybrid logging approach between our log storage and traditional logging.
- We compare the performance of ChronicleDB with commercial, open-source and academic systems in our experiments using real event data.

The remainder of this paper is structured as follows: Section 2 discusses related work and proposes related solutions. In Section 3, we give a brief overview of the system architecture. Section 4 addresses ChronicleDB's storage layout, Section 5 discusses its indexing approach. Recovery issues are examined in Section 6. In Section 7, we evaluate our system experimentally and Section 8 concludes the paper.

2. RELATED WORK

Our discussion of related work is structured as follows. At first, we present data stores relating to ChronicleDB. Then, we discuss previous work referring to our indexing techniques.

Data Stores The domain of ChronicleDB partly relates to different types of storage systems, including data warehouse, event log processing as well as temporal database systems.

One of the first solutions explicitly addressing event data is *DataDepot* [20]. DataDepot is a data warehouse for streaming data, running on top of a relational system. Hence, DataDepot achieves a throughput of only about 10 MiB/s. *Tidalrace* [22], the successor of DataDepot, pursues a distributed storage approach and reaches data rates of up to 500.000 records per second, which still does not compete with ChronicleDB running on a single machine. *DataGarage* [25] is a data warehouse designed for managing performance data on commodity servers which consists of several relational databases stored on a distributed file system. Similar to ChronicleDB, DataGarage addresses aggregation and deletion of outdated events. However, DataGarage is by design a scalable distributed system that is not designed to run as a library tightly integrated in the application code. In addition, DataGarage does not address high ingestion rates.

The most popular NoSQL systems in the context of storing events are Cassandra [1] and HBase [3]. As shown in our experimental section, ChronicleDB clearly outperforms Cassandra when running on a central system.

A representative for log storage systems is *LogBase* [32], which is also applied for event log processing. In contrast to our approach, LogBase is designed as a general-purpose database, also applicable for media data like photos. LogBase is based on HDFS [2] and simply writes data to logs. The authors use an index similar to B^{link}-trees, augmented with compound keys (key, timestamp) to index the data in an in-memory multi-version index.

LogKV [16] utilizes distributed key-value stores to process event log files. In fact, Cassandra [1] was used as underlying key-value store. In experiments, the authors achieved a throughput of 28K events/s ingestion bandwidth per worker node, each consisting of an Intel Xeon X5675 system with 96GB memory and a 7200rpm SAS drive, connected via a 1GB/s network. In comparison to ChronicleDB, the ingestion rate is lower by about a factor of 100.

The third class of storage solutions ChronicleDB partly relates to is that of time series databases. A representative of this class is *tsdb* [18]. Similar to our approach, the authors use a LZ compression for loss-less data compression. In contrast to our approach, time series databases (including tsdb) usually assume that data arrives at every tick.

Gorilla [29] proposes a main-memory time series system on top of HBase with support for ad-hoc query processing. The authors propose a compression technique for uni-variate events of continuous event data.

OpenTSDB [8] and KairosDB [6] are time series database systems on top of HBase and Cassandra. Thus, they have the same deficiencies as their underlying systems.

The mostly related storage system is *InfluxDB* [5], a new open-source time series database solution. As will be discussed in our experimental section, the performance and functionality of InfluxDB on a central system are inferior to ChronicleDB.

Indexing Aggregation in the context of temporal databases has been extensively investigated before in the database community. Widom et al. [34] proposed the SB-tree for partial temporal aggregates. The SB-tree shares some common characteristics with our indexing approach TAB^+-tree. But unlike our approach, a SB-tree only maintains the aggregates for a certain attribute.

More recent research concentrates on observational data and event data. The recently proposed CR-index by Wang et al. [33] is based on LogBase [32] and also utilizes temporal correlation of data. It maintains a separate index per attribute on its minimum/maximum intervals within data blocks. But instead of creating a separate index for each attribute, ChronicleDB keeps all secondary information within a single index. The cost for writing events is lower when the event is written once. In addition, queries on multiple attributes do not need to access multiple indexes.

3. SYSTEM ARCHITECTURE

This section introduces the general architecture of ChronicleDB. At first, we present our requirements on the system. Afterwards, its main components are introduced. Finally, we discuss the main features of ChronicleDB and its fundamental design principles.

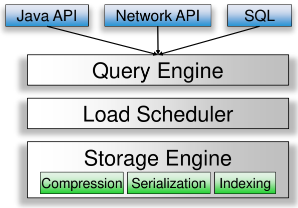


Figure 1: Layers of the ChronicleDB architecture.

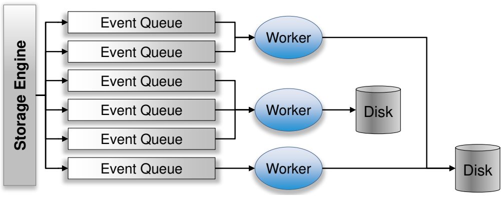


Figure 2: Example of a ChronicleDB topology.

3.1 Requirements

ChronicleDB aims at supporting temporal-relational events, which consist of a timestamp t and several non-temporal, primitive attributes a_i. So, sequences of events (*streams*) can be considered as multi-variate time series, but with non-equidistant timestamps. Timestamps can either refer to system time (when the event occurred at the system) or application time (when the event occurred in the application). The latter is more meaningful to temporal queries on the application level and thus our goal is to maintain a physical order on application time.

Our main objective is fast writing in order to keep-up with high and fluctuating event rates. ChronicleDB should be as economical as possible in order to store data for the long-term, i.e., months or years. Therefore, we aimed at a centralized storage system for cheap disks running as an embedded storage solution within a system (e.g., a self-driving car).

Generally, we assume events to arrive chronologically. Events are inserted into the system **once** and are (possibly) deleted once. In the mean time, there are no updates on an event. However, we also want to support occasional out-of-order insertions as they typically occur in event-based applications [13]. They can happen, e.g., if sensors are sending their events in batches based on asynchronous clocks or simply due to communication problems.

The most important types of queries the system has to support are *time travel queries* and *temporal aggregation queries*. Time travel queries allow requests for specific points and ranges in time, e.g., *all ssh login attempts within the last hour*. Temporal aggregation queries give a comprehensive overview of the data, e.g., *the average number of ssh logins for each day of the week during the last three months*. In addition, the system should efficiently support queries on non-temporal attributes, e.g., *alls ssh logins within the last day from a certain IP range*.

3.2 Architecture Overview

Figure 1 depicts a high level view of ChronicleDB's architecture. In this paper, we focus on the lower layer, i.e., the storage engine and the indexing capabilities of ChronicleDB. Nevertheless, we also give a short description of the other layers, which will be discussed in more detail in future work.

The storage engine of ChronicleDB logically consists of three components: event queues, workers and disks. Basically, *event queues* have two functions. Primarily, they decouple the ingestion of events from further processing. As a side effect, they also compensate chronologically out-of-order event insertion. The *workers* are responsible for writing data to disks and therefore reside in their own threads. Each worker processes the events from its assigned event queues, as long as they are non-empty. All events of a stream are separately stored on one of the worker's dedicated disks.

The architecture of ChronicleDB is sufficiently flexible to take into account workload characteristics as well as the available system resources. The task of the load scheduler is to determine the configuration settings. Figure 2 shows an exemplary topology for seven streams, three workers and two disks.

3.3 Design Principles & System Features

In ChronicleDB, the major design principle is: *the log is the database*. We avoid costly additional logging as the considered append-only scenario does not cause costly random I/Os and therefore does not incur buffering strategies with no-force writes. Only in case of out-of-order arrival of events, we have to deviate from this paradigm as we want to keep the data ordered with respect to application time.

ChronicleDB is implemented in Java and is integrated into the JEPC event processing platform [21]. It supports an embedded as well as a network mode. ChronicleDB offers a high-performance storage solution for event data while supporting load-adaptive indexing and efficient removal of outdated events.

To improve storage utilization as well as write performance, ChronicleDB makes use of (lossless) compression. The main objective is write-optimization, thus we focused on fast compression with reasonable compression rate. Hence, we chose LZ4 [7] as compression algorithm, but any other would be possible.

For data access, the query engine of ChronicleDB supports an SQL-like query language. Additionally, queries can also be processed via a Java API.

4. STORAGE LAYOUT

This section presents the storage layout of ChronicleDB. At first, Section 4.1 describes the problem statement. Section 4.2 introduces the components of the storage layout. Finally, we present the overall layout.

4.1 Problem Statement

The main objective of the storage layout was to support both full sequential write performance and full sequential read performance. Furthermore, we aimed for reasonable performance for random reads and fast recovery support in case of system failures. The challenge was to support these requirements with compressed, i.e., variable-sized data.

We decided to use blocks as compression unit, see Section 4.2.1. Naïvly, the physical offset within a file could be used as identifier (ID) for block addressing. In case of fixed block sizes, the position of a block could be easily computed. But due to compression, blocks are of variable size. So, the final physical offset cannot be computed in advance, which also causes a rethink of the indexing architecture, see Section 5.2 and 5.3.

Therefore, a smart address mapping was required while offering low storage overhead. To the best of our knowledge, there is no existing solution that satisfies these requirements. For example, TokuFS [19] proposes a compressed file system for micro data, but only reaches about 35% of sequential disk speed in the presented experiments. So, we developed a novel storage layout that meets our requirements while offering fast recovery.

4.2 Components

The management of compressed blocks is closely related to that of variable-length records in database systems. They usually manage variable-length records in blocks of fixed size and maintain pointers to the different records within the block. To solve the problem of addressing variable-sized blocks, we adopt this approach. We introduce logical IDs (representing virtual addresses) and an abstraction layer that maps logical IDs to physical addresses. While this solution is quite obvious, the challenging aspect is the storage of the mapping. A straight-forward approach would be to store the physical mapping information *logical IDs → physical addresses* separately. Unfortunately, this incurs random writes and therefore results in a significant performance loss, as we will show in our experimental evaluation. Thus, we decided to store the mapping information *interleaved* with the data.

4.2.1 Blocks

The smallest operational unit of the proposed storage layout is a *logical block (L-block)*. Because we utilize disks as primary storage, we align the L-block size at the size of a physical disk block. Each L-block has to be separately accessible via a unique ID. This is why we chose L-blocks as unit for compression. While L-blocks are of fixed size, the size of a *compressed block* (in the remainder of the paper denoted as *C-block)* depends on its individual compression ratio and therefore, C-blocks are of variable size.

In terms of compression, the optimal physical data storage layout would be a column layout. In terms of write performance, a row layout is superior. We optimized our storage layout for better compression rates utilizing a hybrid approach. ChronicleDB stores relational events in a column-based fashion *only within a single L-block*, similar to the PAX layout [12]. Thus, all data belonging to the same row is organized within the same L-block. At the same time, the column-based ordering of the data within a L-block groups values that are expected to be very similar, which allows better compression.

4.2.2 Macro Blocks

C-blocks are managed in groups of blocks, denoted as *macro blocks*, with a fixed physical size. Nevertheless, the number of C-blocks contained in a macro block varies, depending on the compression rate of the corresponding L-blocks. Macro blocks provide the smallest granularity for physical writes to disk. The size of the macro block has to be a multiple of the L-block size. We impose this constraint for recovery purposes, as will be discussed in detail in Section 6.

Each macro block stores the number of C-blocks it contains as well as the size of each C-block. If a C-block does not completely fit into the current macro block, the C-block is split and the overflow is written to a new macro block. So, macro blocks are dense by default. However, out-of-order events cause updates on C-blocks. In case of dense blocks such updates result in costly macro block overflows. In order to avoid these cost, we reserve a certain amount of spare space for updates in macro blocks. Section 5.7 provides more details on spare space in blocks. Figure 3 shows the layout of a macro block.

4.2.3 Translation Lookaside Buffer

Translation of virtual to physical memory addresses is a fundamental task in computer systems, typically conducted in hardware in the memory management unit (MMU).

In ChronicleDB, we followed a similar approach, but used a software-based *translation lookaside buffer* (TLB). In ChronicleDB, a virtual address is the ID of an L-block. The physical address is the position of the corresponding C-block in the storage layout. The physical address of a C-block c is represented by a tuple (mb_c, p_c), consisting of the position of the corresponding macro block mb_c and the offset p_c within mb_c.

The IDs of L-blocks are simply consecutive numbers. Hence, the TLB only has to store the physical addresses, while the virtual addresses are implicitly given by the position. This TLB structure is related to the CSB$^+$-tree [31], which also uses implicit child pointers to improve the cache behavior of the B$^+$-tree.

The mapping information for recent blocks is kept in memory. Though, to support fast recovery, we have to write parts of the mapping information frequently to disk. Therefore, TLB entries are also managed in blocks, denoted as TLB-blocks. The size of a TLB-block is equal to the size of an L-block. If a TLB-block is filled, it is written to disk. Each TLB-block contains the same amount of entries. E.g., for an L-block size of 8 KiB and 64 bit address size, a TLB-block can contain up to 1020 entries (considering meta data). To support a large address space, we organize TLB-blocks hierarchically in a tree. The resulting TLB tree does not require explicit routing information for address lookup.

Algorithm 1 outlines the address lookup. It starts at the root of the TLB tree, which is always and solely kept in memory. Thanks to the consecutive ID numbering, the index of the corresponding child entry can be easily calculated as well as the associated address. This address is used to load the child block from disk. The algorithm proceeds with the next levels until a leaf node is reached and the final C-block address can be looked up. To speed up address translation, we use a write buffer for each level of the TLB. Furthermore, at least the index levels of the TLB are kept in memory to improve read performance. This should be possible as the size of the TLB index (without the leaf level) is N/b^2 for N C-blocks and L-block size b.

4.3 Overall Layout

The storage layout is designed to avoid random I/Os. Therefore, macro blocks and TLB-blocks are stored interleaved.

In a first possible solution, from the query processing perspective, a TLB-block with k mapping entries should ideally address its k succeeding C-blocks. Unfortunately, this requires either to buffer these k blocks with the risk of data loss in case of a system failure, or to perform a random I/O

```
Algorithm 1: TLB-block Address Lookup
  Input  : ID $id$ of the requested block, entries per
           TLB-block $b$ and TLB height $l$
  Output: The physical address of the C-block
  $index \leftarrow \left\lfloor \frac{id}{b^l} \right\rfloor \mod b$;
  for $i = l - 1$ to 1 do
      $address \leftarrow TLB_{i+1}[index]$;
      load $TLB_i$ from $address$;
      $index \leftarrow \left\lfloor \frac{id}{b^i} \right\rfloor \mod b$;
  end
  return $TLB_0[id \mod b]$
```

to write the TLB-block after writing the k C-blocks. So, there is a tradeoff between performance and safety issues.

In a second solution, we solve this problem by placing the TLB-block behind the data it refers to. So, a TLB-block with k entries always refers to its immediately **preceding** k C-blocks. In this way, we do not have to buffer C-blocks during ingestion, but still avoid random I/Os for writing the mapping information. The drawback of the second solution is that read operations now cause random I/Os. To avoid these random I/Os, a sliding read buffer of k L-blocks is used when a sequential scan is performed. This requires less than 8 MiB memory in case of 8 KiB per L-block. In comparison to the first solution, this approach requires the same amount of buffering, but avoids possible data loss. So, we opted for the second solution.

5. ON INDEXING EVENTS

In this section we present our indexing approach to support the queries we listed in Section 3.1. Among those are time-travel queries, temporal aggregation queries and filter queries on non-temporal attributes.

The remainder of this section is structured as follows: At first, Section 5.1 describes the key characteristics of temporal data. Section 5.2 presents our primary index, secondary indexes are addressed in Section 5.3. Removal of old data is explained in Section 5.4. In Section 5.5, we propose our temporal partitioning and load scheduling approach. Section 5.6 explains how the indexes can efficiently support the targeted types of queries. Finally, Section 5.7 addresses our solution for dealing with out-of-order events.

5.1 Temporal Correlation

In general, we observe in event processing that values occurring within a small time interval are often very similar. Sensor values, e.g., representing temperature or main memory consumption, typically do not change tremendously within short time periods. We call this *temporal correlation.* In agreement with [16], we introduce a formal notation of temporal correlation in the following. For a given sequence A of attribute values $a_i, 1 \leq i \leq N$, we define the average distance as

$$dist(A) := \frac{1}{N-1} \sum_{k=2}^{N} \mid a_k - a_{k-1} \mid$$

This sum is the arithmetic mean of the Manhattan distance. The temporal correlation (tc) is then 1 minus the average distance divided by the range of values within the sequence A. Thus,

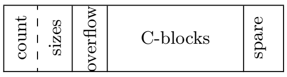


Figure 3: Macro block layout.

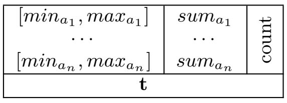


Figure 4: Index entry layout of TAB$^+$-tree.

$$tc(A) := 1 - \frac{dist(A)}{max(A) - min(A)}$$

The value of temporal correlation is in the unit interval. If close to 1, there is a high correlation within the sequence A. We will leverage temporal correlation for lightweight-indexing to speed-up queries.

5.2 TAB$^+$-tree

As primary index for ChronicleDB, we propose the *Temporal Aggregated B$^+$-tree (TAB$^+$-tree).* The TAB$^+$-tree is based on the B$^+$-tree and uses the events' timestamp as key. As usual for B$^+$-trees, the TAB$^+$-tree node size matches block size, i.e., L-block size.

5.2.1 Index Layout

For query processing and recovery issues, we use a linking *in both directions* at *every level* of the tree. We utilize this linking to speed-up query processing as well as to enhance recovery, discussed in Section 6.

To improve query processing, we leverage temporal correlation of event data. For every node in the TAB$^+$-tree, we store the minimum and maximum (min_{a_i},max_{a_i}) value of each attribute A_i. Figure 4 shows the index entry layout.

These min-max values are used for supporting filter queries on non-temporal attributes without the need of an index on the secondary attribute. This approach is called lightweight indexing as it is inexpensive to offer. However, the indexing quality largely depends on the temporal correlation of attributes.

In addition to minimum and maximum values, the TAB$^+$-tree also maintains the *sum* as well as the number of entries (*count*) for each attribute in a subtree. These simple statistics are stored in the index entries, next to the timestamp (t), which represents the key in the TAB$^+$-tree. The storage overhead is very small because aggregates are only maintained in the index levels and the number of attributes is negligible compared to the number of entries in an index node.

5.2.2 Tree Construction

The problem of storing chronological events in an indexed fashion on disks can be solved with an efficient sort-based bulk-loading strategy for B^+-trees. Figure 5 sketches the index construction. Due to the key's sorted nature, the index can be built in *from-left-to-right* fashion while holding the tree's right flank in memory. Because sorting is not required, the cost for index creation is reduced from $O(\frac{N}{b}\ log_b\ \frac{N}{b})$ to $O(\frac{N}{b})$ for N events and block size b. Hence, index construction is almost *for free.* We avoid the traversal of the right flank for each event and build the tree from bottom to top. When a leaf node is filled, its corresponding index entry is inserted into the parent node. Therefore, the parent node

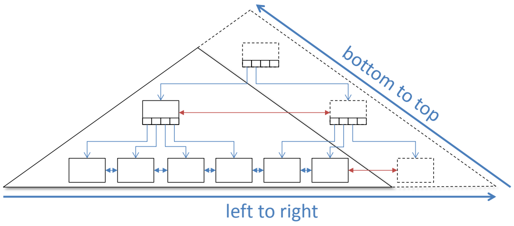


Figure 5: TAB$^+$-tree construction.

has to be accessed only once per child node, which also applies to the entire tree.

A problem arises due to the next neighbor linking (indicated by red arrows in Figure 5). The next neighbor reference has to be known in advance when the node is written to disk. Otherwise, the node would have to be updated later, resulting in random I/Os which would deteriorate the system performance notably. This issue is intensified by data compression. Therefore, stable IDs are necessary as we have discussed in Section 4 already.

5.3 Secondary Indexes

To efficiently support queries on non-temporal attributes *without* high temporal correlation, ChronicleDB also provides secondary indexes. We chose log-structured indexes as they are designed for high write-throughput. Nevertheless, secondary indexes incur high overheads. Hence, ChronicleDB's load scheduler temporally deactivates secondary indexing in case of peak loads, as will be discussed in Section 5.5.

The most popular log-structured index used, e.g., in HBase [3] or TokuDB [23], is the LSM-tree [28]. In addition to LSM trees ChronicleDB also supports cache oblivious look-ahead arrays (COLA), another log-structured index. The advantage of COLA in comparison to a native LSM-tree is its better support for proximity and range queries. To speed-up exact-match queries, we utilize Bloom filters [15], which can be maintained very efficiently.

5.4 Time-Splitting

ChronicleDB is a hybrid between OLTP and OLAP database. In terms of data ingestion, ChronicleDB is like a traditional OLTP system, but queries to ChronicleDB are similar to OLAP queries. Aggregation queries on (historical) data are essential in OLAP systems and commonly address predefined time ranges, like the sales within the last week or month ([17]). ChronicleDB offers the possibility to align data organization to the specific query pattern. Therefore, we introduce *regular time-splits*. After a user-defined amount of time, a new TAB$^+$-tree residing in a separate file is created. E.g, a salesman is interested in weekly sales statistics, so he would choose weeks as regular time split granularity. The same takes place for each secondary index such that the regular time-split covers a fixed interval for all indexes. The regular time-splits are managed within a TAB$^+$-tree again. This enables aggregation queries even in constant time.

Though ChronicleDB aims at long-term storage, it also addresses deletion and reduction of ancient data. Removing events from the TAB$^+$-tree could be realized via cutting off its left flank. However, this would result in costly I/O operations, as the data has to be removed event-by-event. Even more critical: secondary indexes have to be kept consistent. Thus, all events contained in the left flank of the TAB$^+$-tree also have to be removed from the secondary index. Instead of removing data event-by-event, ChronicleDB supports the removal of outdated events at the granularity of regular time splits. Thus, only the corresponding files have to be deleted (logically). Alternatively, outdated events can be thinned out or condensed via aggregation, leveraging the aggregates in the TAB$^+$-tree again.

Regular time-splits enable ChronicleDB to keep local statistics for each time-split. Especially the temporal correlation is an important metric that can be considered to decide which secondary indexes should be maintained. If the temporal correlation for the last split is above a certain threshold, ChronicleDB can switch to lightweight indexing only. This results in systematic partial indexing. Furthermore, time splits allow for higher insertion performance while building secondary indexes compared to one large index. This also has been observed in [27]. Ancient data is removed from ChronicleDB in whole regular time splits, indicated in Figure 6 before rs_1.

5.5 Partial Indexing

Fluctuating data rates are always a very challenging problem, especially in the context of sensor data. We addressed this problem with load scheduling to ensure maximum ingestion speed. In times of moderate input rates, we try to maintain as many secondary indexes as possible. We give higher priority to those indexes on attributes with low temporal correlation. More advanced strategies are possible taking into account the access frequency of the attributes. But in case of a system overload, the load scheduler stops building secondary indexes for attributes with high temporal correlation until ChronicleDB can handle the input again. This results in (unplanned) partial indexes, which have to be synchronized with the primary index again. Therefore, we introduce a second kind of time split, termed *irregular split.*

Figure 6 shows an example where regular splits are prefixed with rs_x, irregular splits with is_x. For each time split, P_x denotes a TAB$^+$-tree, S_x a secondary index. At is_3, secondary indexation has been switched off due to a system overload. Therefore, the primary index is split, too. If the system load decreases, secondary indexes are switched on again. Re-activation only takes place at regular splits. In this example, secondary indexation continues after rs_5. In case of sufficient resources, ChronicleDB can also rebuild secondary indexes for previous time splits that emerged during an overload, e.g., $[is_3, rs_4]$ as well as $[rs_4, rs_5]$.

5.6 Query Processing

In this section we discuss how a TAB$^+$-tree can be utilized for query processing.

5.6.1 Time travel queries

Due to its descent from B$^+$-tree and the fact that events are indexed based on their (start) timestamps, the TAB$^+$-tree performs a time travel query like a range query in a B$^+$-tree. The leaf nodes of the TAB$^+$-tree can be sequentially traversed from left to right using the linking between adjacent leaf nodes until an event occurs that is outside of the temporal query range. Thus, the total cost is logarithmic (in the number of events) plus the time to access the

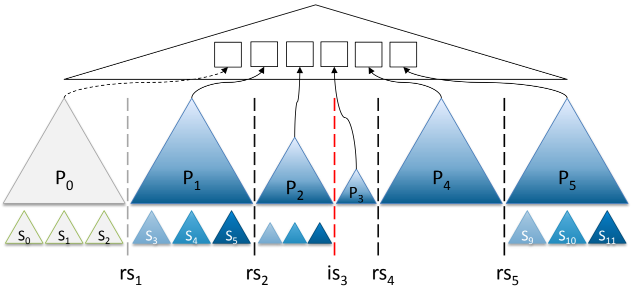


Figure 6: Index scheduling example.

required leaves.

In addition, we also utilize restrictions on non-temporal attributes specified in the query to speed-up query processing. Then, the (min, max) information of the corresponding attributes is used for pruning. If the query interval for a specific attribute a and the interval (min_a, max_a) of a TAB$^+$-tree node are disjoint during tree traversal, the node is skipped.

Algorithm 2: TAB$^+$-tree pruning query

```
Input   : Time interval $[t_s, t_e]$, range $[min_{a_i}, max_{a_i}]$ for
          attributes $A_i$
Stack s, Node n, Index i;
s.push(root);
s.push(0);
while !s.isEmpty() do
    n ← s.pop();
    i ← s.pop();
    while i < n.size do
        if n.isLeaf() then
            if n [i ].$t > t_e$ then
                return; /* No further results */
            else if n [i ].$t \geq t_s$ then
                output n [i ];
            end
        else if Intersection (n [i ],intervals) then
            s.push(n);
            s.push(i +1);
            n ← n [i ].child;
            i ← 0;
            continue;
        else if i > 0 AND n [i-1].$t > t_e$ then
            return; /* No further results */
        end
        i ← i +1;
    end
end
```

5.6.2 *Temporal aggregation queries*

Temporal aggregation queries compute an aggregation value (sum, avg, stdev, count, min, max) for a given point or range in time. Again, the TAB$^+$-tree acts as guide for the temporal dimension. Additionally, the aggregation information per node can be utilized. If a node in the TAB$^+$-tree is fully covered by the query range, ChronicleDB can exploit the node's aggregate value (covering all of its child nodes). If the node's time interval is intersected by the given query range, the query proceeds with its qualifying child nodes. Therefore, also temporal aggregation queries are answered in logarithmic time.

5.6.3 *Secondary queries in TAB$^+$-tree*

Secondary queries can utilize the inherent lightweight indexing of the TAB$^+$-tree. Algorithm 2 sketches the secondary query processing. As input, the requested time interval as well as the restrictions on the desired attributes are provided. For simplicity, the proposed algorithm reports all query results; in fact, query processing in ChronicleDB is demand-driven. The TAB$^+$-tree is traversed in depth-first order by means of a stack while nodes are pruned as early as possible. The stack keeps track of the current tree path as well as the index of the last visited entry for each tree level.

5.7 Managing out-of-order Data

So far, we have assumed an unexceptional chronological order of the incoming events. This is, however, not satisfied in real scenarios where asynchronous clocks, network delays and faulty devices cause exceptional out-of-order arrival of events. This problem is well-known in event processing [13], but also has a serious impact on the design of ChronicleDB. There are two basic solutions for dealing with out-of-order arrivals of events. First, we could change the notion of time in the TAB$^+$-tree. Instead of using application time as the primary attribute for indexing, we could use system time. By definition, the events are then always in correct order because an event item receives its timestamp at arrival in ChronicleDB. Furthermore, application time should be used as an additional attribute indexed in a lightweight fashion within the TAB$^+$-tree. This causes additional cost in query processing, in particular for aggregate queries. The second solution still maintains the TAB$^+$-tree as an index on application time (as described in the previous sections). We will pursue this approach in the following.

5.7.1 *Out-of-order Buffers*

In order to deal with out-of-order, we introduce the Algorithm 3 that is illustrated in Figure 7. First, we try to insert incoming out-of-order events into the right flank buffer of the tree. If the timestamp of an event is too far in the past, we insert the event into a dedicated queue sorted with respect to application time. When this queue becomes full, we flush its entries *in bulk* into the TAB$^+$-tree. To prevent data loss in case of a system crash, all events in the queue are additionally written to a mirror log in system time order.

While the sorted queue serves to leverage temporal locality, we also target at leveraging physical locality in the storage layout. Therefore, an additional buffer in combination with a write-ahead log [26] and no-force write strategy is introduced for the TAB$^+$-tree .

Without any further modifications, this approach would still cause serious problems in the TAB$^+$-tree. First of all, an out-of-order insertion will often hit a full L-block. Consequently, a split would be triggered and the sequential layout would be damaged causing higher costs for queries. In order to avoid these splits, we propose to reserve a certain amount of spare space in an L-block for absorbing out-of-order insertions without structural modifications. This is only meaningful if the number of out-of-order arrivals is not extremely high. For example, if we expect 15 out-of-order

Algorithm 3: Out-of-order Insertion

```
Input: Out-of-order event $e$, right flank of TAB$^+$-tree
       $flank$, sorted queue $queue$, mirror log $log$
if $e.timestamp > flank[1].getLast().timestamp$ then
    // Add $e$ to leaf of the TAB$^+$-tree flank
    $flank[0].add(e)$;
else
    // Add $e$ to sorted queue
    $queue.add(e)$;
    // Write e to the mirror log
    $log.append(e)$;
    if $queue.isFull()$ then
        // Flush all events from $queue$
        for $qe$ in $queue$ do
            insert $qe$ from bottom to top into $flank$;
        end
        Clear $log$;
    end
end
```

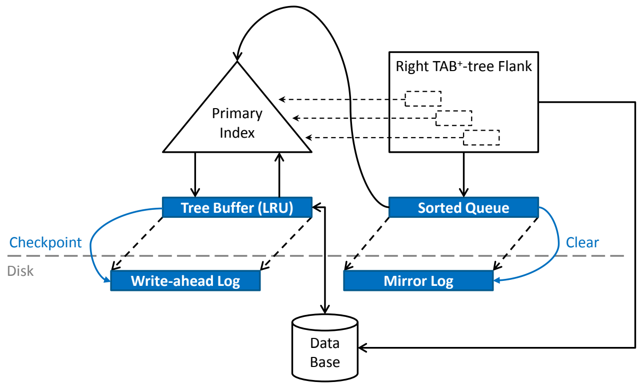


Figure 7: Buffer layout and data flow for out-of-order data.

events per L-block, a simple urn-based analysis shows that the probability of an overflow is less than 10% for a spare space of 20 events.

In addition, we also address additional spare space on the storage level. Each L-block corresponds to a compressed C-block which size depends on the compression rate. A reduction of the compression rate results in an increased C-block size. For example, an update on an aggregate could lead to an increased C-block size, even though the L-block size has not changed. Thus, macro blocks also reserve a certain amount of spare space. If a C-block exceeds the remaining spare space of its macro block, it is moved to the end of the database and a reference entry is written at its original position.

5.7.2 *Keeping Secondary Indexes consistent*

References in the secondary index are represented by physical addresses. So, references to relocated C-blocks in the storage layout will become invalid in a secondary index. One solution for this problem is to update the affected references in the secondary index. However, since there can be many secondary indexes, eager reference updates are very expensive.

Thus, we use the following lazy approach instead where a split of a block does not trigger an update of the entries in the secondary indexes. In order to maintain the search capabilities of secondary indexes, we store the timestamp of the event in the corresponding index entry of a secondary index. In addition, a flag in each block is kept for indicating whether a block is split or not. If a search via a secondary index arrives at a block that has been split, we use the timestamp to search for the event in the primary index. For all other blocks we still use the direct linkage.

6. FAILURES AND RECOVERY

This section addresses the recovery capabilities of ChronicleDB after a system crash. Recovery takes place in three steps. At first, the storage layout is recovered. Subsequently, the primary index is restored and finally the logs are processed to transfer the system into a consistent state again.

6.1 Storage Layout Recovery

In ChronicleDB, the most critical part of the storage layout is its address translation, i.e., the TLB. As the root and the right flank of the TLB are only kept in memory, any information about block address translation is lost in case of a system crash. Rebuilding the TLB would require a full database scan. However, this is not acceptable for a database we expect to be very large (in the range of terabytes).

In order to support fast reconstruction of the TLB's right flank, we introduce references within the TLB. As only the recently created part of the TLB has to be restored, recovery is performed from the end of the database to its start. Each TLB-block keeps a reference to its previous TLB-block on the same level. Given the last successfully written TLB-block, its predecessor can be directly accessed. The recovery has to scan all TLB-blocks that are children of the last (and therefore lost) TLB-block in the parent level. Then, recovery continues with the next level. To support the direct access to upper levels, TLB-blocks additionally store a reference to its parent's predecessor TLB-block. Thus, these references implicitly create checkpoints for each level. Figure 8 shows the linking of TLB-blocks. For presentation purposes, we assume two address entries per TLB-block. For example, the leaf d_{10} is a child of m_1 and keeps an extra pointer to d_9 that is the predecessor of m_1.

The TLB recovery is outlined in Algorithm 4. In case of a crash, the last written TLB-block is seeked on disk. This is simple as the size of a macro block is a multiple of TLB-block size. There are two possibilities for the classification of the last written L-block: either it is a TLB-block or it is part of a macro block. In the latter case, the previous L-block is read until the last successfully written TLB-block is found. The upper bound for the number of L-blocks to be read before finding a TLB-block is the number of entries per TLB-block. After having located the last successfully written TLB-block the recovery of the TLB continues by leveraging the introduced references.

The predecessors of the last written TLB-blocks are located until the parent reference is different. The corresponding references of these TLB-blocks (except the last) are used to rebuild the parent node. The recovery continues at the next upper level with the parent's previous entry. At each level of the TLB, the number of entries to be read is limited by the number of entries per TLB-block.

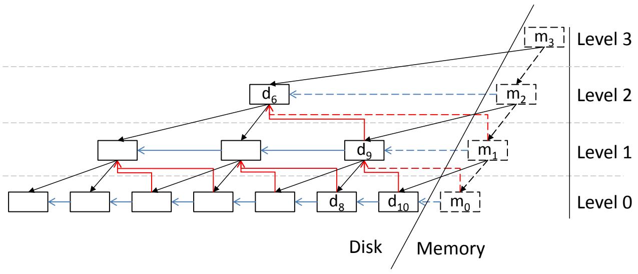


Figure 8: TLB structure with recovery references.

Figure 8 gives an example. In case of a system failure, the TLB-blocks $m_0 - m_3$ are lost. The recovery starts to discover d_{10} first, which was referenced in m_1 before the crash. Its predecessor, d_8, has a different previous parent reference. So, recovery continues with d_9 and afterwards with d_6. After that, $m_1 - m_3$ are recovered. m_0 is restored by simply scanning all macro blocks after d_{10}.

Algorithm 4: TLB Recovery

```
Input: Database size in bytes $s$, L-block size in bytes $b$
$b_{addr} \leftarrow \left\lfloor \frac{s}{b} \right\rfloor * b$ ;   // Last complete block address
$block \leftarrow read(b_{addr})$;
// Lookup last TLB-block
while $block$ is not a TLB-block do
    $b_{addr} \leftarrow b_{addr} - b$;
    $block \leftarrow read(b_{addr})$;
end
// Rebuild TLB
while $block.prev$ != $null$ do
    $prev \leftarrow read(block.prev)$ ; // Read previous entry
    if $prev.prevParent$ != $block.prevParent$ then
        // Switch to the next higher TLB level
        $block \leftarrow read(block.prevParent)$;
    else
        // Restore the reference in the new
           parent entry
        Add $b_{addr}$ to $TLB_{block.level+1}$;
    end
end
```

6.2 TAB$^+$-tree Recovery

In the second step of the system recovery, the right flank of the TAB$^+$-tree is reconstructed. This reconstruction is very similar to the TLB recovery and starts scanning the data base in reverse order for the last successfully written TAB$^+$-tree node. After locating the last node n_i at level i, a new index entry is inserted into the tree's right flank at level $i + 1$. In the next step, all nodes of level i belonging to the same parent node are iterated utilizing the previous neighbor linking at all levels of the TAB$^+$-tree. The recovery continues recursively with the last written node of the parent level until the root is reached.

6.3 Log Recovery

Finally, the consistency of the data base has to be ensured. This step only matters in case of previously occurred out-of-order events. At first, the write-ahead log is processed from start to end. For each log entry, its LSN is compared with the LSN of the block it refers to. If the LSN of the block is smaller than the entry's LSN, the associated event is regularly inserted into the TAB$^+$-tree. Finally, the sorted queue is restored by scanning the mirror log.

7. EXPERIMENTAL EVALUATION

This section presents a selection of important results from an extensive performance comparison. Section 7.1 describes the experimental setup, Section 7.2 evaluates the storage layout of ChronicleDB and Section 7.3 evaluates the query performance. Section 7.4 compares ChronicleDB with open-source (Cassandra), commercial (InfluxDB) academic systems (LogBase in combination with CR-index). Finally, Section 7.5 investigates the performance impact of out-of-order data.

7.1 Experimental Setup

All experiments were conducted on a Windows 7 desktop computer with Intel I7 2600 quad-core CPU at 3.4 GHz and 8 GB DDR3 RAM, equipped with an 1 TB HDD and 128 GB SSD. The latter is only used for writing the out-of-order logs. We run various experiments to identify the impact of parameters on the performance of ChronicleDB and to chose the best settings. The L-block size and the size of macro blocks are two parameters we set to 8 KiB and 32 KiB, respectively. Smaller block sizes (e.g. 4 KiB) as well as larger block sizes (e.g. 32 KiB) perform slightly inferior to our standard settings. Because we measured only a minor impact of these parameters, we do not detail these results. Unless specified otherwise, the experiments with ChronicleDB where conducted with 10 % spare for an L-block and without partial indexing on a single worker.

In our experiments we used four data sets termed CDS, BerlinMod, DEBS and SafeCast. *CDS* is a synthetic data set with eight numerical attributes and a timestamp. This data set was generated based on real-world cpu data [14]. *DEBS* is a real data set, extracted from the DEBS Grand Challenge 2013 data [11]. The data provides sensor readings of a soccer game. We used the data set obtained from the ball. *BerlinMOD* is a semi-synthetic data set, sampled from a collection of taxi trips in Berlin. We used the pre-calculated trips data available at [4]. *SafeCast* [10] contains spatio-temporal radiation data collected by the community. We extracted spatial and temporal attributes as well as the radiation. Table 1 reports important properties: the number of events, the size of an event, the compression rate, the minimum temporal correlation among all attributes of the corresponding data set and the time for reading the input into memory. As these data sets are ordered by time, they are not suited for out-of-order experiments. We will postpone the generation of out-of-order data to Section 7.5.

7.2 Compression and Recovery

First of all, we evaluate the performance of the storage layout presented in Section 4, in the following denoted as *ChronicleDB layout*. We compare ChronicleDB layout with a completely separated storage layout (*separate layout*), storing the address information of the blocks from the data of the TAB$^+$-tree in a separate file.

In order to evaluate the storage layout, we measured the impact of compression. We run experiments with a hy-

Table 1: Indicators of the data sets.

Data set	#Events	Bytes/Event	Compression	minimum tc	Input Processing (s)
DEBS	24,278,210	76	34.37%	0.476	53.14
BerlinMOD	56,129,943	48	71.14 %	0.9996	285.655
SafeCast	40,193,450	36	64.08 %	0.9622	354.093
CDS	20,000,000	72	68,36 %	0.869	0.618

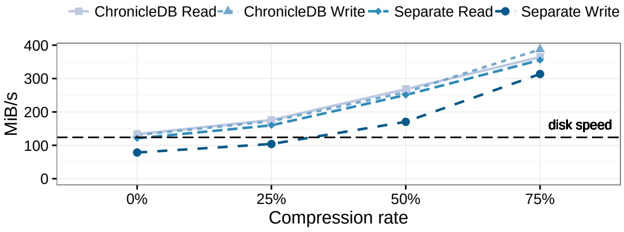


Figure 9: Throughput as a function of the compression rate for different storage layouts.

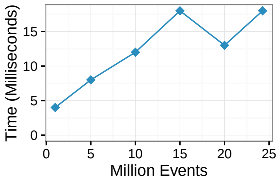


Figure 10: TLB recovery time after ingesting various numbers of events.

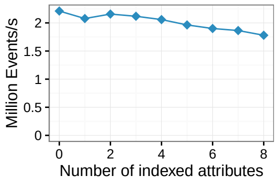


Figure 11: Write throughput as a function of the number of indexed attributes.

pothetical compression rate that is constant for all blocks. Figure 9 shows that the write as well as the read performance of ChronicleDB layout scales almost linearly with the compression rate. In addition, we measured the sequential disk speed by writing data without address information and without compression. This results in 123.89 MiB/s that matches sequential disk speed. Without compression ChronicleDB achieves almost the same results, while the write performance of the separate layout drops to 71.59 MiB/s. This shows the advantage of ChronicleDB layout where data blocks and TLB blocks are kept interleaved in a single file.

Finally, we discuss the recovery times of ChronicleDB layout. Therefore, we triggered a system crash after ingesting a predefined number of events from DEBS and measured the recovery time for the TLB. The results are depicted as a function of the number of ingested events in Figure 10. Note that recovery of the storage layout requires only a few milliseconds, independent of the number of events. The recovery time is not a perfect monotonic function because it is determined by the fill degree of the nodes from the right flank of the TLB.

7.3 Query Performance

At first, we discuss the TAB$^+$-tree lightweight indexing performance for the data set CDS. Therefore, we report the impact of the number of (lightweight) indexed attributes on the overall ingestion performance, depicted in Figure 11. There is a very mild linear performance decrease in the number of indexed attributes because of the capacity reduction of internal nodes in the TAB$^+$-tree.

7.3.1 Time-Travel & Temporal Aggregation Queries

Next, we discuss the query times for time-travel queries as well as temporal aggregation queries in ChronicleDB while varying the temporal range (selectivity). We used the DEBS data set in this experiment. Figure 12 depicts the total processing time as a function of selectivity. The performance of the time travel queries decreases linear in the selectivity, while the logarithmic performance of the aggregate query seems to be constant.

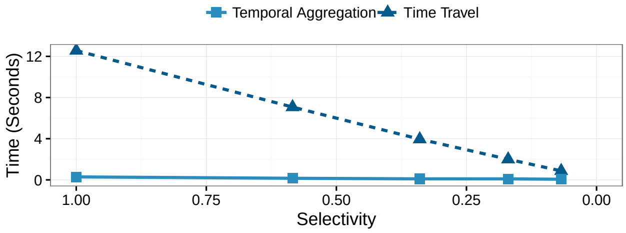
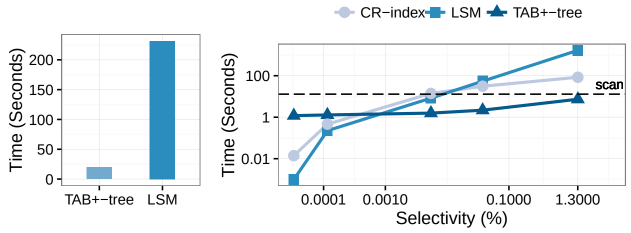


Figure 12: Performance evaluation of time-travel queries and temporal aggregation queries on DEBS.

7.3.2 Secondary Indexes

Finally, we evaluate the index capabilities of ChronicleDB. Therefore, we ingested the DEBS data set into ChronicleDB twice. First, we used lightweight indexing (TAB$^+$-tree) on *velocity*, the attribute with smallest temporal correlation. In the second built, we used a secondary index (LSM-tree) on the same attribute. As shown in Figure 13a, the build time is substantially higher in case a LSM-tree is generated.

Figure 13b presents our query results for ChronicleDB with TAB$^+$-tree and LSM-tree respectively (note the log-scale). Additionally, we compared the results with the CR-index in LogBase. Therefore, we used the configuration from [33] and deployed LogBase on the local file system of the same machine as ChronicleDB. We also depict the time for a full range scan in ChronicleDB as a dashed line. In summary, LogBase with CR-index is inferior to ChronicleDB. For very low selectivity, the secondary LSM index in ChronicleDB performs best, slightly better than the CR-index. In contrast to CR-index, TAB$^+$-tree is not fully kept in memory, which explains the lower query performance for very low selectivities. In case of higher selectivities, the TAB$^+$-tree is significantly faster than both LSM and CR-Index. In case of LSM, the low temporal correlation of *velocity* introduces many random accesses resulting in poor query performance. To find the break-even in query performance between LSM and TAB$^+$-tree, the selectivity as well as the temporal correlation have to be taken into account. But due to the high



(a) Loading

(b) Queries

Figure 13: Secondary index evaluation on DEBS in LogBase (CR-index) and in ChronicleDB with lightweight index (TAB$^+$-tree) and with secondary index (LSM).

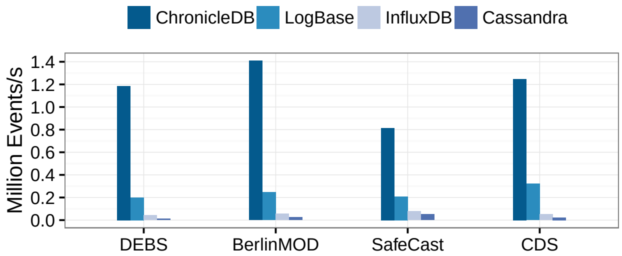


Figure 14: Ingestion throughput benchmark.

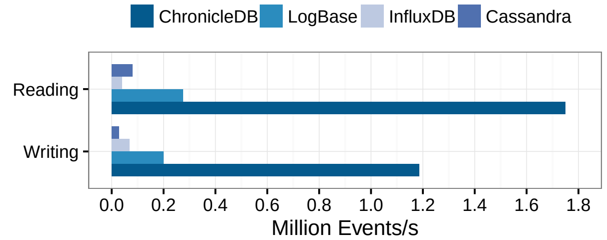


Figure 15: Write and read throughput comparison with LogBase, Cassandra and InfluxDB on DEBS.

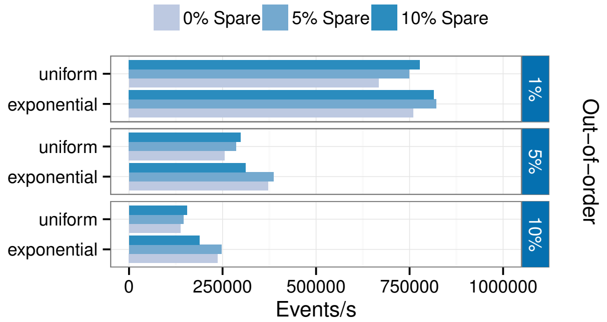


Figure 16: Out-of-order ingestion performance

cost for index creation, the LSM is only justified for highly read-intensive applications.

7.4 Benchmarking ChronicleDB

In the following, we compare ChronicleDB with sInfluxDB (v0.9), Cassandra (v2.0.14) and LogBase, all running on the same machine as ChronicleDB. In terms of write-performance as well as read throughput, Cassandra is currently one of the fastest representatives of distributed key-value stores (see Rabl et al. [30]). For InfluxDB we used batches of 5K events and a batch interval of one second to reduce network overhead. Cassandra does not offer batches for performance improvement. We used the JAVA client libraries for InfluxDB[1] and Cassandra[2]. For LogBase, we applied the suggested configuration from [33] again. Figure 14 reports the throughput of the four systems for our data sets. We did not include here the time for reading and converting the input data, see Table 1. In summary, ChronicleDB clearly outperforms Cassandra, InfluxDB and Logbase. In case of CDS, ChronicleDB is superior to Cassandra and InfluxDB by a factor of 50 and 22, respectively. For LogBase, the speedup is still more than a factor of three.

We also report the performance of full relation scans (exemplary for DEBS), as replaying of historical data is an important feature of a historical data store. In case of InfluxDB, we used only half of the data due to limitations regarding the response size of a query. As presented in Figure 15, ChronicleDB outperforms LogBase by a factor of 5, Cassandra and InfluxDB by a factor of 22 and 43, respectively.

7.5 Out-of-order Data

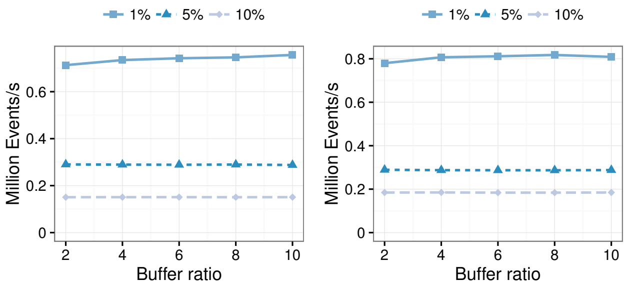
In order to examine the out-of-order insertion performance, we modified the timestamps of the CDS data as follows. Out-of-order insertions take place in bulk after every 10K insertions of chronological events. The delay of out-of-order data is restricted to the time interval since the last out-of-order bulk insertion, simulating late arrivals from a sensor. We consider two distributions for a delay: uniform and exponential. For an exponential distribution, smaller delays occur more often than longer ones (with an expected delay of 40 ms).

Figure 16 shows the results of our experiments with different fractions of out-of-order data as well as varying amounts of spare for uniform and exponential delay distribution. Out-of-order inserts are expensive. The throughput for 10% out-of-order is smaller by a factor of three than that of 1%. Nevertheless, even for an out-of-order rate of 10%, ChronicleDB outperforms InfluxDB by more than an order of magnitude. As expected, exponential distribution performs slightly better during ingestion because of higher locality in the buffer. The read performance is very similar for all approaches at about 1.4M events per second. In general, sparing improves ingestion performance as well as read performance because larger reorganizations can be avoided and there is no need to remap blocks.

We also measured the influence of the ratio between the range of out-of-order data and the buffer size, depicted in Figure 17. For example, a buffer ratio of 2 indicates that the

[1] https://github.com/influxdb/influxdb-java

[2] https://github.com/datastax/java-driver



(a) Uniform (b) Exponential

Figure 17: Evaluation of the buffer ratio impact.

buffer covers half of the out-of-order data. The size of the out-of-order buffer does not have a significant influence on the overall performance, as the system is CPU-bound due to overheads for compression and serialization.

8. CONCLUSION & FUTURE WORK

The design of a database system for event streams is nowadays important to tackle the very high ingestion rates in new application related to IoT. Not all of these applications allow large-scale distributed database systems, but require a tightly integrated database solution in the application code. In this paper, we presented ChronicleDB, a new type of centralized database system that exploits the temporal arrival order of events. We discussed in detail its storage management, indexing support and recovery capabilities. Due to its dedicated system design, our experimental results showed a great superiority of ChronicleDB in comparison to distributed systems like Cassandra and InfluxDB that are widely used for write-intensive applications like the management of event streams.

So far, we put our focus on the careful design of ChronicleDB as a centralized system and showed that simply making standard systems scalable is not the right answer. However, there is no reason for not using ChronicleDB in a distributed environment. In our current and future work, we examine how to exploit the benefits of distributed frameworks for write-intensive applications. We even believe that ChronicleDB's write-once policy and its storage layout suits well to distributed file systems like HDFS.

9. REFERENCES

[1] Apache Cassandra. http://cassandra.apache.org/.

[2] Apache Hadoop. http://hadoop.apache.org/.

[3] Apache HBase. http://hbase.apache.org/.

[4] BerlinMOD. http://dna.fernuni-hagen.de/secondo/BerlinMOD/BerlinMOD.html.

[5] InfluxDB. https://github.com/influxdata/influxdb.

[6] KairosDB. https://kairosdb.github.io/.

[7] LZ4 Compression. https://code.google.com/p/lz4/.

[8] OpenTSDB. http://opentsdb.net/.

[9] PostgreSQL. http://www.postgresql.org/.

[10] SafeCast. http://blog.safecast.org/data/.

[11] DEBS Grand Challenge 2013. http://www.orgs.ttu.edu/debs2013/index.php?goto=cfchallengedetails, 2013.

[12] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, pages 169–180, 2001.

[13] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, pages 363–374, 2007.

[14] L. Baumgärtner, C. Strack, B. Hoßbach, M. Seidemann, B. Seeger, and B. Freisleben. Complex Event Processing for Reactive Security Monitoring in Virtualized Computer Systems. In *DEBS*, pages 22–33, 2015.

[15] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.

[16] Z. Cao, S. Chen, F. Li, M. Wang, and X. S. Wang. LogKV: Exploiting Key-Value Stores for Log Processing. In *CIDR*, 2013.

[17] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[18] L. Deri, S. Mainardi, and F. Fusco. Tsdb: A compressed database for time series. In *TMA*, pages 143–156, 2012.

[19] J. Esmet, M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. The TokuFS Streaming File System. In *HotStorage*, pages 14–14, 2012.

[20] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk. Stream Warehousing with DataDepot. In *SIGMOD*, pages 847–854, New York, NY, USA, 2009. ACM.

[21] B. Hoßbach, N. Glombiewski, A. Morgen, F. Ritter, and B. Seeger. JEPC: The Java Event Processing Connectivity. *Datenbank-Spektrum*, 13(3):167–178, 2013.

[22] T. Johnson and V. Shkapenyuk. Data Stream Warehousing in Tidalrace. In *CIDR*, 2015.

[23] B. Kuszmaul. How TokuDB Fractal Tree Indexes Work. Technical report, TokuTek, 2010.

[24] P. L. Lehman and s. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, Dec. 1981.

[25] C. Loboz, S. Smyl, and S. Nath. DataGarage: Warehousing Massive Performance Data on Commodity Servers. *PVLDB*, 3(1-2):1447–1458, 2010.

[26] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[27] P. Muth, P. O'Neil, A. Pick, and G. Weikum. The LHAM Log-structured History Data Access Method. *The VLDB Journal*, 8(3-4):199–221, 2000.

[28] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[29] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: A Fast, Scalable, In-memory Time Series Database. *PVLDB*, 8(12):1816–1827, 2015.

[30] T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, and S. Mankovskii. Solving Big Data Challenges for Enterprise Application Performance Management. *PVLDB*, 5(12):1724–1735, 2012.

[31] J. Rao and K. A. Ross. Making B+- Trees Cache Conscious in Main Memory. In *SIGMOD*, pages 475–486, 2000.

[32] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. LogBase: A Scalable Log-structured Database System in the Cloud. *PVLDB*, 5(10):1004–1015, 2012.

[33] S. Wang, D. Maier, and B. C. Ooi. Lightweight Indexing of Observational Data in Log-Structured Storage. In *PVLDB*, volume 7, pages 529–540, 2014.

[34] J. Yang and J. Widom. Incremental Computation and Maintenance of Temporal Aggregates. *The VLDB Journal*, 12(3):262–283, 2003.

EXstream: Explaining Anomalies in Event Stream Monitoring

Haopeng Zhang, Yanlei Diao, Alexandra Meliou
College of Information and Computer Sciences, University of Massachusetts Amherst
{haopeng, yanlei, ameli}@cs.umass.edu

ABSTRACT

In this paper, we present the EXstream system that provides high-quality explanations for anomalous behaviors that users annotate on CEP-based monitoring results. Given the new requirements for explanations, namely, *conciseness*, *consistency with human interpretation*, and *prediction power*, most existing techniques cannot produce explanations that satisfy all three of them. The key technical contributions of this work include a formal definition of optimally explaining anomalies in CEP monitoring, and three key techniques for generating sufficient feature space, characterizing the contribution of each feature to the explanation, and selecting a small subset of features as the optimal explanation, respectively. Evaluation using two real-world use cases shows that EXstream can outperform existing techniques significantly in conciseness and consistency while achieving comparable high prediction power and retaining a highly efficient implementation of a data stream system.

1. INTRODUCTION

Complex Event Processing (CEP) extracts useful information from large-volume event streams in real-time. Users define interesting patterns in a CEP query language (e.g,. [3, 4]). With expressive query languages and high performance processing power, CEP technology is now at the core of real-time monitoring in a variety of areas, including the Internet of Things [16], financial market analysis [16], and cluster monitoring [26].

However, today's CEP technology supports only *passive monitoring* by requesting the monitoring application (or user) to explicitly define patterns of interest. There is a recent realization that many real-world applications demand a new service beyond passive monitoring, that is, the ability of the monitoring system to identify interesting patterns (including anomalous behaviors), produce a concrete explanation from the raw data, and based on the explanation enable a user action to prevent or remedy the effect of an anomaly. We broadly refer to this new service as *proactive monitoring*. We present two motivating applications as follows.

1.1 Motivating Applications

Production Cluster Monitoring. Cluster monitoring is crucial to many enterprise businesses. For a concrete example, consider a production Hadoop cluster that executes a mix of Extract-Transform-Load (ETL) workloads, SQL queries, and data stream tasks. The programming model of Hadoop is MapReduce, where a MapReduce job is composed of a `map` function that performs data transformation and filtering, and a `reduce` function that performs aggregation or more complex analytics for all the data sharing the same key. During job execution, the map tasks (called mappers) read raw data and generate intermediate results, and the reduce tasks (reducers) read the output of mappers and generate final output. Many of the Hadoop jobs have deadlines because any delay in these jobs will affect the entire daily operations of the enterprise business. As a result, monitoring of the progress of these Hadoop jobs has become a crucial component of the business operations.

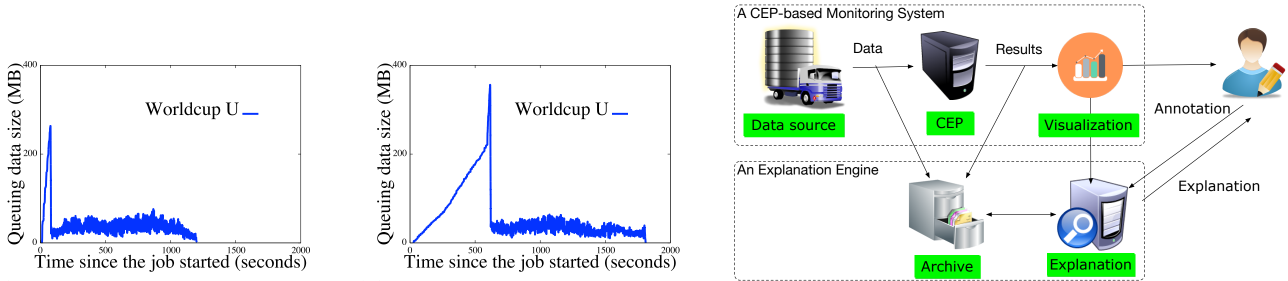
However, the Hadoop system does not provide sufficient monitoring functionality by itself. CEP technology has been shown to be efficient and effective for monitoring a variety of measures [26]. By utilizing the event logs generated by Hadoop and system metrics collected by Ganglia[12], CEP queries can be used to monitor Hadoop job progress; to find tasks that cause cluster imbalance; to find data pull stragglers; and to compute the statistics of lifetime of mappers and reducers. Consider a concrete example below, where the CEP query monitors the size of intermediate results that have been queued between mappers and reducers.

Example 1.1 (Data Queuing Monitoring)**.** Collect all the events capturing intermediate data generation/consumption for each Hadoop job. Return the accumulative intermediate data size calculated from those events (Q1).

Figure 1(a) shows the data queuing size of a monitored Hadoop job. The X-axis stands for the time elapsed since the beginning of the job, while the Y-axis represents the size of queued data. In this case, the job progress turns out to be normal: the intermediate results output by the mappers start to queue at the beginning and reach a peak after a short period of time. This is because a number of mappers have completed in this period while the reducers have not been scheduled to consume the map output. Afterwards, the queued data size decreases and then stabilizes for a long period of time, meaning that the mappers and reducers are producing and consuming data at constant rates, until the queued data reduces to zero at the end of the job.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.15



(a) Data queuing size of a normal Hadoop job

(b) Data queuing size of an abnormal Hadoop job

(c) Architecture of EXstream

Figure 1: Hadoop cluster monitoring: examples and system architecture.

Suppose that a Hadoop user sees a different progress plot, as shown Figure 1(b), for the same job on another day: there is a long initial period where the data queuing size increases gradually but continually, and this phase causes the job completion time to be delayed by more than 500 seconds. When the user sees the job with an odd shape in Figure 1(b), he may start considering the following questions:

- What is happening with the submitted job?
- Should I wait for the job to complete or re-submit it?
- Is the phenomenon caused by the bugs in the code or some system anomalies?
- What should I do to bring the job progress back to normal?

Today's CEP technology, unfortunately, does not provide any additional information that helps answer the above questions. The best practice is manual exploration by the Hadoop user: he can dig into the complex Hadoop logs and manually correlate the Hadoop events with the system metrics such as CPU and memory usage returned by a cluster monitoring tool like Ganglia [12]. If he is lucky to get help from the cluster administrator, he may collect additional information such as the number of jobs executed concurrently with his job and the resources consumed by those jobs.

For our example query, the odd shape in Figure 1(b) is due to high memory usage of other programs in the Hadoop cluster. However, this fact is not obvious from the visualization of the user's monitoring query, Q1. It requires analyzing additional data beyond what is used to compute Q1 (which used data relevant only to the user's Hadoop job, but not all the jobs in the system). Furthermore, the discovery of the fact requires new tools that can automatically generate explanations for the anomalies in monitoring results such that these explanations can be understood by the human and lead to corrective / preventive actions in the future.

Supply Chain Management. The second use case is derived from an aerospace company with a global supply chain. By talking with the experts in supply chain management, we identified an initial set of issues in the company's complex production process which may lead to imperfect or faulty products. For instance, in the manufacturing process of a certain product the environmental features must to be strictly controlled because they affect the quality of production. For example, the temperature and humidity need to be controlled in a certain range, and they are recorded by the sensors deployed in the environment. However, if some sensors stop working, the environmental features may not be controlled properly and hence the products manufactured during that period can have quality issues. When such anomalies arise, it is a huge amount of work to investigate the claims from customers given the complexity of manufacturing process and to analyze a large set of historical data to find explanations that are meaningful and actionable.

1.2 Problem Statement and Contributions

The overall goal of EXstream is to provide good explanations for anomalous behaviors that users annotate on CEP monitoring results. We assume that an enterprise information system has CEP monitoring functionality: a CEP monitoring system offers a dashboard to illustrate high-level metrics computed by a CEP query, such as job progress, network traffic, and data queuing. When a user observes an abnormal value in the monitoring results, he annotates the value in the dashboard and requests EXstream to search for an explanation from the archived raw data streams. EXstream generates an optimal explanation(formalized in Section 2.2) by quickly replaying a fraction of the archived data streams. Then the explanation can be encoded into the system for proactive monitoring for similar anomalies in the future.

Challenges. The challenges in the design of XStream arise from the requirements for such explanations. Informed by the two real-world applications mentioned above, we consider three requirements in this work: (a) *Conciseness*: The system should favor smaller explanations, which are easier for humans to understand. (b) *Consistency*: The system should produce explanations that are consistent with human interpretation. In practice, this means that explanations should match the true reasons for an anomaly (*ground truth*). (c) *Prediction power*: We prefer explanations that have predictive value for future anomalies.

It is difficult for existing techniques to meet all three requirements. In particular, prediction techniques such as logistic regression and decision trees [2] suffer severely in conciseness or consistency as shown in our evaluation results. This is because these techniques were designed for prediction, but not for explanations with conciseness and consistency requirements. Recent database research [25, 20] seeks to explain outliners in SQL query answers. This line of work assumes that explanations can be found by searching through various subsets of the tuples that were used to compute the query answers. This assumption does not suit real-world stream systems for two reasons: As shown for our example, Q1, the explanation of memory usage contention among different jobs cannot be generated from only those events that produced the monitoring results of Q1. Furthermore, the stream execution model does not allow us to

Event type	Meaning	Schema
JobStart	Recording a Hadoop job starts	(timestamp, eventType, eventId, jobId, clusterNodeNumber)
JobEnd	Recording a Hadoop job finishes	(timestamp, eventType, eventId, jobId, clusterNodeNumber)
DataIO	Recording the activities of generation (positive values) / consumption (negative values) of intermediate data	(timestamp, eventType, eventId, jobId, taskId, attemptId, clusterNodeNumber, dataSize)
CPUUsage	Recording the CPU usage for a node in the cluster	(timestamp, eventType, eventId, clusterNodeNumber, CPUUsage)
MemUsage	Recording the memory usage for a node in the cluster	(timestamp, eventType, eventId, clusterNodeNumber, memUsage)

Figure 2: Example event types in Hadoop cluster monitoring. Event types can be specific to the Hadoop job (e.g., JobStart, DataIO, JobEnd), or they may report system metrics (e.g., CPUUsage, FreeMemory).

Q	Pattern SEQ($Component_1$, $Component_2$, . . .) Where [$partitionAttribute$] $\wedge$ $Pred_1$ $\wedge$ $Pred_2$ $\wedge$ . . . Return ($timestamp$, $partitionAttribute$, $derivedA_1$, $derivedA_2$, . . .)[]	Q_1	Pattern SEQ(JobStart a, DataIO+ b[], JobEnd c) Where [jobId] Return ($b[i]$.timestamp, a.jobId, $sum(b[1\cdots i]$.dataSize))[]

Figure 3: Syntax of a query in SASE (on the left), and an example query for monitoring data activity (on the right).

repeat query execution over different subsets of events or perform any precomputation in a given database [20].

Contributions. In this work, we take an important step towards discovering high-quality explanations for anomalies observed in monitoring results. Toward this goal, we make the following contributions:

1) Formalizing explanations (Section 2): We provide a formal definition of optimally explaining anomalies in CEP monitoring as a problem that maximizes the information reward provided by the explanation.

2) Sufficient feature space (Section 3): A key insight in our work is that discovering explanations first requires a sufficient feature space that includes all necessary features for explaining observed anomalies. EXstream includes a new module that automatically transforms raw data streams into a richer feature space, **F**, to enable explanations.

3) Entropy-based, single-feature reward (Section 4): As a basis for building the information reward of an explanation, we model the reward that each feature, $f \in \mathbf{F}$, may contribute using a new entropy-based distance function.

4) Optimal explanations via submodular optimization (Section 5): We next model the problem of finding an optimal explanation from the feature space, **F**, as a submodular maximization problem. Since submodular optimization is NP-hard, we design a heuristic algorithm that ranks and filters features efficiently and effectively.

5) Evaluation (Section 6): We have implemented EXstream on top of the SASE stream engine [3, 26]. Experiments using two real-world use cases show promising results: (1) Our entropy distance function outperforms state-of-the-art distance functions on time series by reducing the features considered by 94.6%. (2) EXstream significantly outperforms logistic regression [2], decision tree [2], majority voting [15] and data fusion [19] in consistency and conciseness of explanations while achieving comparable, high predication accuracy. Specifically, it outperforms others by improving consistency from 10.7% to 87.5% on average, and reduces 90.5% of features on average to ensure conciseness. (3) Our implementation is also efficient: with 2000 concurrent monitoring queries, the triggered explanation analysis returns explanations within half a minute and affects the performance only slightly, delaying events processing by 0.4 second on average.

2. EXPLAINING CEP ANOMALIES

The goal of EXstream is to provide good explanations for anomalous behaviors that users annotate on CEP-based monitoring results. We first describe the system setup, and give examples of monitoring queries and anomalous observations that a user may annotate. We then discuss the requirements for providing explanations for such anomalies, and examine whether some existing approaches can derive explanations that fit these requirements. Finally, we define the problem of optimally explaining anomalies in our setting.

2.1 CEP Monitoring System and Queries

In this section, we describe the system setup for our problem setting. The overall architecture of EXstream is shown in Figure 1(c). Within the top dashed rectangle in Figure 1(c) is a CEP-based monitoring system. We consider a data source S, generating events of n types, $\mathbf{E} = \{E_1, E_2, \ldots, E_n\}$. Events of these types are received by the CEP-based monitoring system continuously. Each event type follows a schema, comprised of a set of attributes; all event schemas share a common timestamp attribute. The timestamp attribute records the occurrence time of each event. Figure 2 shows some example event types in the Hadoop cluster monitoring use case [26].

We consider a CEP engine that monitors these events using user-defined queries. For the purposes of this paper, monitoring queries are defined in the SASE query language [3], but this is not a restriction of our framework, and our results extend to other CEP query languages. Figure 3 shows the general syntax of CEP queries in SASE, and an example query, Q_1, from the Hadoop cluster monitoring use case. Q_1 collects all data-relevant events during the lifetime of a Hadoop job. We now explain the main components of a SASE query.

Sequence. A query Q may specify a sequence using the `SEQ` operator, which requires components in the sequence to occur in the specified order. One component is either a single event or the Kleene closure of events. For example, Q_1 specifies three components: the first component is a single event of the type *JobStart*; the second component is a Kleene closure of a set of events of the type *DataIO*; and the third component is a single event of type *JobEnd*.

Predicates. Q can also specify a set of predicates in its `Where` clause. One special predicate among these is the bracketed *partitionAttribute*. The brackets apply an equivalence test on the attribute inside, which requires all selected events to have the same value for this attribute. The *partitionAttribute* tells the CEP engine which attribute to partition by. In Q_1, *jobId* is the partition attribute.

Return matches. Q specifies the matches to return in the `Return` clause. Matches comprise a series of events with raw or derived attributes; we assume *timestamp* and the *partitionAttribute* are included in the returned events. We denote with m a match on one partition and with M_Q the set of all matches. Q_1 returns a series of events based on selected *DataIO* events, and the returned attributes include *timestamp*, *jobId*, and a derived attribute— the total size for all selected *DataIO* events. In order to visualize results in real time, matches will be sent to the visualization module as events are collected.

Visualizations and feedback. Our system visualizes matches from monitoring queries on a dashboard that users can interact with. The visualizations typically display the (relative) occurrence time on the X-axis. The Y-axis represents one of the derived attributes in returned events. Users can specify simple filters to focus on particular partitions. All returned events of M_Q are stored in a relational table T_{M_Q}, and the data to be visualized for a particular partition is specified as $\pi_{t,attr\_i}(\sigma_{partitionAttribute=v}(M))$. Figure 1(a) shows the visualization of a partition, which corresponds to a Hadoop job for this query. In this visualization, the X-axis displays the time elapsed since the job started, and the Y-axis shows the derived sum over the "DataSize" attribute.

Users can interact with the visualizations by annotating anomalies. For example, the visualization of Figure 1(b) demonstrates an unexpected behavior, with the queueing data size growing slowly. A user can drag and draw rectangles on the visualization, to annotate the abnormal component, as well as reference intervals that demonstrate normal behavior. We show an example of these annotations in Figure 4. A user may also annotate an entire period as abnormal, and choose a reference interval in a different partition. The annotations will be sent to the explanation engine of EXstream, which is shown in the bottom dashed rectangle of Figure 1(c). The explanation engine will be introduced in detail in following sections. We use I_A to denote the annotated abnormal interval in a partition P_A: $I_A = (Q, [lower, upper], P_A)$. We use I_R to denote the reference interval, which can be explicitly annotated by the user, or inferred by EXstream as the non-annotated parts of the partition. We write $I_R = (Q, [lower, upper], P_R)$, where P_R and P_A might be the same or different partitions.

2.2 Explaining Anomalies

Monitoring visualizations allow users to observe the evolution of various performance metrics in the system. While the visualizations help indicate that something may be unusual (when an anomaly is observed), they do not offer clues that point to the reasons for the unexpected behavior. In our example from Figure 4, there are two underlying reasons for the abnormal behavior: (1) the free memory is lower than normal, and (2) the free swap space is lower than normal. However, these reasons are not obvious from the visualization; rather, a Hadoop expert had to manually check a large volume of logs to derive this explanation. Our goal is to automate this process, by designing a system that seamlessly integrates with CEP monitoring visualizations, and which can produce explanations for surprising observations.

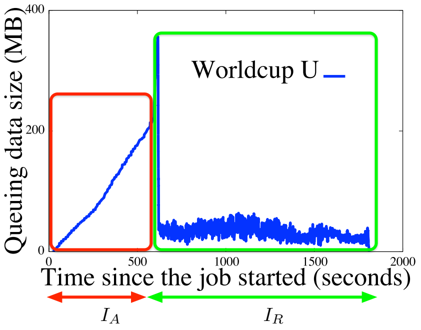


Figure 4: Abnormal (I_A) and reference (I_R) intervals.

We define three desirable criteria for producing explanations in EXstream:

1. **Conciseness**: The system should favor smaller, and thus simpler explanations. Conciseness follows the Occam's razor principle, and produces explanations that are easier for humans to understand.
2. **Consistency**: The system should produce explanations that are consistent with human interpretation. In practice, this means that explanations should match the true reasons for an anomaly (*ground truth*).
3. **Prediction power**: We prefer explanations that have predictive value for future anomalies. Such explanations can be used to perform *proactive monitoring*.

Explanations through predictive models. The first step of our study explored the viability of existing prediction techniques for the task of producing explanations for CEP monitoring anomalies. Prediction techniques typically learn a model from training data; by using the anomaly and reference annotations as the training data, the produced model can be perceived as an explanation. For now, we will assume that a sufficient set of features is provided for training (we discuss how to construct the feature space in Section 3), and evaluate the explanations produced by two standard prediction techniques for the example of Figure 4.

Logistic regression [2] produces models as weights over a set of features. The algorithm processes events from the two annotated intervals as training data, and the trained prediction model — a classifier between abnormal and reference classes — can be considered an explanation to the anomaly. The resulting logistic regression model for this example is shown in Figure 5. While the model has good predictive power, it is too complex, and cannot facilitate human understanding of the reported anomaly. The model assigns non-zero weights to 30 out of 345 input features, and while the two ground truth explanations identified by the human expert are among these features (23 and 24), their weights in this model are low. This model is too noisy to be of use, and it is not helpful as an explanation.

No.	Feature	Weight
1	DataIOFrequency	-0.01376
2	CPUIdleMean	0.0089
3	PullFinishFrequency	-0.00708
4	ProcTotalMean	0.00085
...	...	...
23	SwapFreeMean	-4.79E-07
24	MemFreeMean	-3.28E-07
...	...	...
30	BoottimeMean	2.61E-10

Figure 5: Model generated by logistic regression for the annotated anomaly of Figure 4.

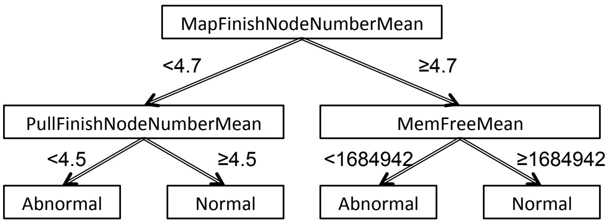


Figure 6: Model Generated by Decision Tree

Decision tree [2] builds a tree for prediction. Each non-leaf node of the tree is a predicate while leaf nodes are prediction decisions. Figure 6 shows the resulting tree for our example. The decision tree algorithm selects three features for the non-leaf nodes, and only one of them is part of the ground truth determined by our expert. The other two features happen to be *coincidentally* correlated with the two intervals, as revealed in our profiling. This model is more concise than the result of logistic regression, but it is not consistent with the ground truth.

The above analyses showed that prediction techniques are not suitable for producing explanations in our setting. While the produced models have good predictive power (as this is what the techniques are designed for), they make poor explanations, as they suffer in consistency and conciseness. Our goal is to design a method for deriving explanations that satisfies all three criteria (Figure 7).

2.3 Formalizing Explanations

Explanations need to be understandable to human users, and thus need to have a simple format. EXstream builds explanations as a conjunction of predicates. In their general format, explanations are defined as follows.

Definition 2.1 (Explanation)**.** An explanation is a boolean expression in Conjunctive Normal Form (CNF). It contains a conjunction of clauses, each clause is a disjunction of predicates, and each predicate is of the form $\{v\ o\ c\}$, where v is a variable value, c is a constant, and o is one of five operators: $o \in \{>, \geq, =, \leq, <\}$.

Example 2.1. The formal form of the true explanations for the anomaly annotated in Figure 4 is ($MemFreeMean$ < 1978482 ∧ $SwapFreeMean$ < 361462), which is a conjunction of two predicates. It means that the average available memory is less than 1.9GB and free swap space is less than 360MB. The two predicates indicate that the memory usage is high in the system (due to resource contention), thus the job runs slower than normal.

Arriving at the explanation of Example 2.1 requires two non-trivial components. First, we need to identify important features for the annotated intervals (e.g., $MemFreeMean$, $SwapFreeMean$); these features will be the basis of forming meaningful predicates for the explanations. Second, we have to derive the best explanation given a metric of optimality. For example, the explanation ($MemFreeMean$ < 1978482) is worse than ($MemFreeMean$ < 1978482 ∧ $SwapFreeMean$ < 361462), because, while it is smaller, it does not cover all issues that contribute to the anomaly, and is thus less consistent with the ground truth.

Algorithm	Conciseness	Consistency	Prediction quality
Logistic regression	Bad	Bad	Good
Decision tree	Ok	Bad	Good
Goal	Good	Good	Good

Figure 7: Performance of prediction methods on our three criteria for explanations.

Ultimately, explanations need to balance two somewhat conflicting goals: simplicity, which pushes explanations to smaller sizes, and informativeness, which pushes explanations to larger sizes to increase the information content. We model these goals through a reward function that models the information that an explanation carries, and we define the problem of deriving optimal explanations as the problem of maximizing this reward function.

Definition 2.2 (Optimal Explanation)**.** Given an archive of data streams D for CEP, a user-annotated abnormal interval I_A and a user-annotated reference interval I_R, an optimal explanation e is one that maximizes a non-monotone, submodular information reward R over the annotated intervals: $\text{argmax}_e\, R_{I_A,I_R}(e)$

The reward function in Definition 2.2 is governed by an important property: rewards are not additive, but *submodular*. This means that the sum of the reward of two explanations is greater than or equal to the reward of their union: $R_{I_A,I_R}(e_1) + R_{I_A,I_R}(e_2) \geq R_{I_A,I_R}(e_1 \cup e_2)$. The intuition for the submodularity property is based on the observation that adding predicates to a conjunctive explanation offers diminishing returns: the more features an explanation already has, the lower the reward of adding a new predicate tends to be. Moreover, R is non-monotone. This means that adding predicates to an explanation *could decrease the reward.* This is due to the conciseness requirement that penalizes big explanations. The optimal explanation problem (Definition 2.2) is therefore a submodular maximization problem, which is known to be NP-hard [11].

2.4 Existing Approximation Methods

Submodular optimization problems are commonly addressed with greedy approximation techniques. We next investigate the viability of these methods for our problem setting.

For this analysis, we assume a reward function for explanations based on mutual information. Mutual information is a measure of mutual dependence between features. This is important in our problem setting, as features are often correlated. For example, $PullStartFrequency$ and $PullFinishFrequency$ are highly correlated, because they always appear together for every pull operation. For this precise reason, Definition 2.2 demands a submodular reward function. Mutual information satisfies the submodularity property. Greedy algorithms are often used in mutual infor-

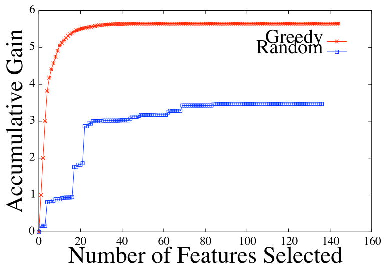


Figure 8: Accumulative mutual information gain under greedy and random strategies.

mation maximization problems. The way they would work in this setting is the following: given an explanation e, which is initially empty, at each greedy step, we select the feature f that maximizes the mutual information of $e \cup f$.

Figure 8 shows the performance of the greedy algorithm for maximization of mutual information, with a strawman alternative. The random algorithm selects a random feature at each step. The greedy strategy clearly outperforms the alternative by reaching higher mutual information gains with fewer features, but it still selects a large number of features (around 20-30 features before it levels off). This means that this method produces explanations that are too large, and unlikely to be useful for human understanding.

2.5 Overview of the EXstream Approach

Since standard approaches for solving the optimal explanation problem are insufficient for our problem setting, we develop a new heuristic method based on good intuitions to address the problem. We next provide a high-level overview of our approach in building EXstream.

1. Sufficient feature space (Section 3): A key insight in our work is that discovering optimal explanations first requires a sufficient feature space that includes all necessary features for explaining observed anomalies. Our work differs fundamentally from existing work on discovering explanations from databases [25, 20]: First, EXstream operates on raw data streams, as opposed to the data carefully curated and stored in a relational database. Second, EXstream does not assume that the raw data streams carry all necessary features for explaining anomalous behaviors. In our above example, the feature, $SwapFreeMean$, captures average free swap space and it does not exist in Hadoop event logs or Ganglia output. Our system includes a module that automatically transforms raw data streams into a richer feature space, $\mathbf{F}$, to enable the discovery of optimal explanations.

2. Entropy-based, single-feature reward (Section 4): As a basis for building the information reward defined in Definition 2.2, we consider the reward that each feature, $f \in \mathbf{F}$, may contribute. To capture the reward in such a base case, we propose a new, entropy-based distance function that is defined on a single feature across the abnormal interval, I_A, and the reference interval, I_R. The larger the distance, the more differentiating power over the two intervals that the feature contributes, and hence more reward produced.

3. Optimal explanations via submodular optimization (Section 5): The next task is to find an optimal explanation from the feature space, $\mathbf{F}$, that maximizes the information reward provided by the explanation. The reward function in Definition 2.2 is non-monotone and submodular, resulting in a submodular maximization problem. Since submodular optimization is NP-hard, our goal is to design a heuristic to solve this problem. Our heuristic algorithm first uses the entropy-based, single-feature reward to rank features, subsequently identifies a cut-off to reject features with low reward, and finally uses correlation-based filtering to eliminate features with information overlap (emulating the submodularity property). Our evaluation shows that our heuristic method is extremely effective in practice.

timestamp	node	usagePercent
4	2	35
5	5	49
6	8	99
7	1	86
8	2	61
9	6	43

Figure 9: Sample events in the type of $CPUUsage$.

3. DISCOVERING USEFUL FEATURES

Explanations comprise of predicates on measurable properties of the CEP system. We call such properties *features*. Some features for our running example are $DiskFreeMean$, $MemFreeMean$, $DataIOFrequency$, etc. In most existing work on explanations, features are typically determined by the query or the schema of the data (e.g., the query predicates in Scorpion [25]). In CEP monitoring, using as features the query predicates or data attributes is not sufficient, because many factors that impact the observed performance are due to other events and changes in the system. This poses an additional challenge in our problem setting, as the set of relevant features is non-trivial. In this section, we discuss how EXstream derives the space of features as a first step to producing explanations.

In an explanation problem, we are given an anomaly interval I_A and a reference interval I_R; the relevant features for this explanation problem are built from events that occurred during these two intervals. To support the functionality of providing explanations, the CEP system has to maintain an archive of the streaming data. The system has the ability to purge archived data after the relevant monitoring queries terminate, but maintaining the data for longer can be useful, as the reference interval can be specified on any past data.

Formally, the events arriving in a CEP system in input streams and the generated matches compose the input to the feature space construction problem. We assume that the CEP system maintains a table for each event type, such as the one depicted in Figure 9. That is, for each event type E_i, logically there is a relational table $R(E_i)$ to store all events of this type in temporal order. There is also a table $R(M)$ to archive all match events, denoted as type M. Let D denote the database for EXstream, which is composed of those tables. So, D is defined as $D = \{R(E_i)|1 \leq i \leq n\} \cup R(M)$.

Each attribute in event type E_i, except the timestamp, forms a time series in a given interval (which can be an anomaly interval I_A or a reference interval I_R). Such time series as features are called *raw features*.

Example 3.1. The table of Figure 9 records events of type $CPUUsage$ in a given time interval [4, 9], and forms two raw features, from two time series. The first one is $CPUUsage.Node$, and its values are ((4, 2), (5,5), (6,8), (7,1), (8,2), (9,6)); the other is $CPUUsage.UsagePercent$ with values ((4,35),

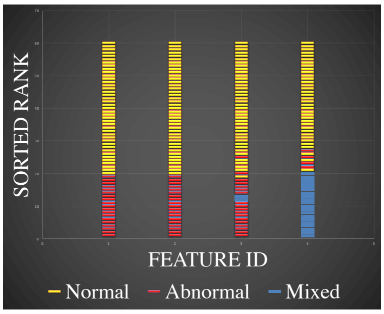


Figure 10: Visualization of the separating power of four features: (1) free memory size, (2) idle CPU percentage, (3) CPU percentage used by IO, and (4) system load. This visualization is not part of EXstream, but we show it here for exposition purposes.

(5,49), (6,99), (7,86), (8,61), (9,43)).

We found that the raw feature space is not good for deriving explanations due to noise. Instead, we need higher-level features, which we construct by applying aggregation functions to features at different granularities. We apply sliding windows over the time series features and over each window, aggregate functions including *count* and *avg* to generate new time serious features. The EXstream system has an open architecture that allows any window size and any new aggregate functions to be used in the feature generation process. Features produced this way are "smoothed" time series; they demonstrate more general trends than raw features, and outliers are smoothed. Example high-level features that we produce by applying aggregations over windows on the raw features are $DataIOFrequency$ and $MemFreeMean$.

4. SINGLE-FEATURE REWARD

In this section, we present the core of our technique: an entropy-based distance function that models the reward of a single feature. We first discuss the intuition and requirements for this function, we then discuss existing, state-of-the-art distance functions and explain why they are not effective in this setting, and, finally, we present our new entropy-based distance metric.

4.1 Motivation and Insights

In seeking explanations for CEP monitoring anomalies, users contrast an anomaly interval with a reference interval. An intuitive way to think about the different behaviors in the two intervals is to consider the differences in the events that occur within each interval. We can measure this difference per feature: how different is each feature between the reference and the anomaly. Each feature is a vector of values, a time series, and our goal is to measure the distance between the time series of a feature during the abnormal interval and the time series of the same feature during the normal interval.

To explain one of the desirable properties of the distance function, we visualize a feature as follows: We order the values of a feature in increasing order and assign a color to each value; red for values that appear in the abnormal interval only, yellow for values that appear in the normal interval only, and blue for values that appear in both normal and abnormal intervals. Figure 10 shows this visualization for 4 different features. In this figure, we note that the first 2 features show a clear separation of values between the normal and abnormal periods. The third feature has less clear separation, but still shows the trend that lower values are more likely to be abnormal. Finally, the fourth feature is mixed for a significant portion of values.

Intuitively, the first two features in Figure 10 are better explanations for the anomaly, and thus have higher reward. The first feature means when the anomalies occur, the free memory size is relatively low, while during the reference interval the free memory size is relatively high. The second feature means that during the abnormal interval, idle CPU percentage is low while it is high during the reference interval. The unclear separation of the other two features, in particular the blue segments, indicate randomness between the two intervals, making them less suitable to explain the annotated anomalies.

This example provides insights on the properties that we need from the distance function: it should favor clear separation of normal and abnormal values, and it should penalize features with mixed segments (values that appear in both normal and abnormal periods). Therefore, the reward of a feature is high if the feature has good separating power, and it is lower with more segmentation in its values.

4.2 Existing State of the Art

Distance functions measuring similarities of time series have been well studied [24], and there is over a dozen distance functions in the literature. However, these metrics were designed with different goals in mind, and they do not fit our explanation problem well. We discuss this issue for the two major categories of distance functions [24].

Lock-step measure. In the comparison of two time series, lock-step measures compare the ith point in one time series to exactly the ith point in another. Such measures include the Manhattan distance (L_1), Euclidean distance (L_2) [9], other L_p-norms distances and approximation based $DISSIM$ distance. Those distance functions treat each pair of points independently, but in our case, we need to compare the time series holistically. For example, assume four simple time series: $TS_1 = (1,1,1)$, $TS_2 = (0,0,0)$, $TS_3 = (1,0,1)$ and $TS_4 = (0,1,0)$. Based on our separating power criterion, $D(TS_1, TS_2)$ should be larger than $D(TS_3, TS_4)$ because there is a clear separation between the values of TS_1 and TS_2, while the values of TS_3 and TS_4 are conflicting. However, applying any of the lock-step measures produces $D(TS_1, TS_2) = D(TS_3, TS_4)$.

Elastic measure. Elastic measures allow comparison of one-to-many points to find the minimum difference between two time series. These measures try to compare time series on overall patterns. For example, Dynamic Time Warping (DTW) tries to stretch or compress one time series to better match another time series; while Longest Common SubSequence(LCSS) is based on the longest common subsequence model. Although these measures also take value difference into account, the additional emphasis on pattern matching makes them ill-suited for our problem.

Both lock-step and elastic measures fall in the category of sequence-based metrics. This means that they consider the order of values. Lock-step functions perform strict step-by-step, or event-by-event comparisons; such rigid measures cannot find similarities in the flexible event series of our problem setting. Elastic measures allow more flexibility, but the emphasis on matching the microstructure of sequences introduces too much randomness in the metric.

In our case, temporal ordering is not important, because we assume the sample points in time series are independent. This makes set-based functions a better fit (as opposed to sequence-based). Set-based functions measure the macro trend while smoothing low-level details.

4.3 Entropy-Based Single-Feature Reward

Since existing distance functions are not suitable to model single-feature rewards, we design a new distance function that emphasizes the separation of feature values between normal and abnormal intervals (Section 4.1). Our distance function is inspired by an entropy-based discretization technique [10], which cuts continuous values into value intervals by minimizing the class information entropy. The segmentation visualized in Figure 10, shows an intuitive connection with entropy: The more mixed the color segments are, the higher the entropy (i.e., more bits are needed to describe the distribution). We continue with some background definitions, and then define our entropy-based distance function, which we will use to model single-feature rewards.

Definition 4.1 (Class Entropy)**.** Class entropy is the information needed to describe the class distributions between two time series. Given a pair of time series, TS_A and TS_R, belonging to the abnormal and reference classes, respectively. Let $|TS_A|$ and $|TS_R|$ denote the number of points in the two time series, let $p_A = \frac{|TS_A|}{|TS_A|+|TS_R|}$, and let $p_R = \frac{|TS_R|}{|TS_A|+|TS_R|}$. Then, the entropy of the class distribution is:

$$H_{Class}(f) = p_A * log(\frac{1}{p_A}) + p_R * log(\frac{1}{p_R}) \quad (1)$$

Definition 4.2 (Segmentation Entropy)**.** Segmentation entropy is the information needed to describe how merged points are segmented by class labels. If there are n segmentations, and p_i represents the ratio of data points included in the ith segmentation, the segmentation entropy is:

$$H_{Segmentation} = \sum_{i=1}^{n} p_i * log(\frac{1}{p_i}) \quad (2)$$

Complicated segmentations in a feature result in more entropy. When there is a clear separation of the two classes, as in the first two features of Figure 10, the segmentation entropy is the same as the class entropy. Otherwise, the segmentation entropy is more than the class entropy.

Penalizing for mixed segments. Segmentation entropy captures the segmentation of the normal and abnormal classes, but does not penalize mixed segments with values that appear in both classes (blue segments in the visualization). Take an extreme case, where all values appear in both classes (single mixed segment). This is the scenario with the worst separation power, but its segmentation entropy is 0, because it is treated as a single segment. This indicates that we need special treatment for mixed (blue) segments.

We assume the worst case distribution of normal and abnormal data points within the segment. This is the uniform distribution, which leads to most segmentation and highest entropy. For example, if a mixed segment c consists of 5 data points, 3 contributed from the normal class (N) and 2 contributed from the abnormal class (A), distributing them uniformly leads to 5 segments: (N,A,N,A,N). We denote this worst-case ordering of segment c as c^*. We assign a penalty term for each segment c, which is equal to the segmentation entropy of its worst-case ordering, c^*: $H_{Segmentation}(c^*)$. We thus define the regularized segmentation entropy:

$$H^+_{Segmentation} = H_{Segmentation} + \sum_{j=1}^{m} H_{Segmentation}(c^*_j) \quad (3)$$

The first term in this formula is the segmentation entropy of the feature, and the second term sums the regularization penalties of all mixed segments (m).

Accounting for feature size. Features may be of different sizes, as different event types may occur more frequently than others. The segmentation entropy is only comparable between two features f_1, f_2, if $|f_1.TS_A| = |f_2.TS_A|$ and $|f_1.TS_R| = |f_2.TS_R|$. However this does not hold for most features. To make these metrics comparable, we normalize segmentation entropy using class entropy and get the following definition for our entropy-based feature distance:

$$D(f) = \frac{H_{Class}(f)}{H^+_{Segmentation}(f)} \quad (4)$$

We use this distance function as a measure of single-feature reward. Features with perfect separation, such as the first two features of Figure 10, have reward equal to 1. Features with more complex segmentation have lower rewards. For the 4 features displayed in Figure 10, the rewards are 1, 1, 0.31, and 0.18, respectively.

5. CONSTRUCTING EXPLANATIONS

The entropy-based single-feature reward identifies the features that best distinguish the normal and abnormal periods. However, ranking the features based on this distance metric is not sufficient to generate explanations. We need to address three additional challenges. First, it is not clear how to select a set of features from the ranked list. There is no specific constant k for selecting a set of top-k features, and moreover, such a set would likely not be meaningful as a top-k set is likely to contain highly-correlated features with redundant information. Second, there are cases where large distances are coincidental, and not associated with anomalies. Third, the rewards are computed for each feature individually, and due to submodularity, they are not additive. Determining how to combine features into an explanation requires eliminating redundancies due to feature correlations.

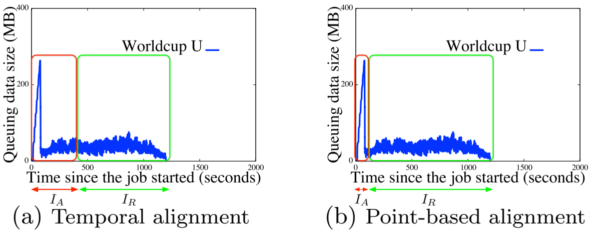
We proceed to describe the EXstream approach to constructing explanations by addressing these challenges in three steps. Each step filters the feature set to eliminate features based on intuitive criteria, until we are left with a high-quality explanation.

5.1 Step 1: reward leap filtering

The single-feature distance function produces a ranking of all features based on their individual rewards. Sharp changes in the reward between successive features in the ranking indicate a semantic change: Features that rank below a sharp drop in the reward are unlikely to contribute to an explanation. Therefore, features whose distance is low, relatively to other features, can be safely discarded.

5.2 Step 2: false positive filtering

It is possible for features to have high rewards due to reasons unrelated to the investigated anomaly. For example, a



(a) Temporal alignment (b) Point-based alignment

Figure 11: Two ways of alignment

feature that measures system uptime can have strong separating power between the annotated anomaly and reference regions (e.g., the anomaly is before the reference), but this is simply due to the nature of the particular feature, and it is not related to the anomaly. We call these features false positives. Our method for identifying and purging such false positives leverages other partitions (e.g., other Hadoop jobs in our running example). The intuition is that if a feature is a false positive, the feature will demonstrate similar behavior in other partitions without an indication of anomaly.

Identifying related partitions. We search the archived streams to identify similar partitions. Intuitively, such partitions should be results generated by the same query, monitoring the same Hadoop program, on the same dataset. EXstream maintains a record of partitions in a partition table to facilitate fast retrieval. The partition table contains dimension attributes that record categorical information (e.g., $CEP - QueryID$, $HadoopJobName$, $Dataset$) about the partition, and measure attributes that record partition statistics (e.g., monitoring duration, number of points). The system identifies related partitions, as those that match the dimension attributes.

Partition alignment. Once it discovers related partitions, EXstream needs to map the annotated regions to each related partition. This alignment can be temporal-based or point-based. In temporal-based alignment, an annotation is mapped to a partition based on its temporal length. For example, in Figure 4, the abnormal period occupies 31% of temporal length; this annotation will align with the the first 31% of the temporal length in a related partition (Figure 11(a)). In point-based alignment an annotation is mapped to a partition based on the ratio of data points that it occupies in the monitoring graph. For example, the annotated high-memory usage partition of Figure 4 includes 113,070 points, with 2116 points falling in the abnormal annotation; this annotation will align with the first equal fraction of points in a related partition (Figure 11(b)). EXstream selects the alignment for which the two partitions have the smallest relative difference. For example, if a related partition has 10% more points, but is 50% longer in time compared to the annotated partition, point-based alignment is preferred.

Interval labeling. Alignment maps the annotations to all related partitions. Now, these new annotations need to be labeled as normal or abnormal. EXstream assigns labels through hierarchical clustering: a period that is placed in the same cluster as the annotated anomaly is labeled as abnormal. The clustering uses two distance functions: entropy-based, and normalized difference of frequencies. Periods whose cluster is far from the anomaly cluster are labeled as normal (reference). Finally, periods that cannot be assigned with certainty are discarded and not used later for validation.

Feature	Reward (annotated)	Reward (all)
Free memory size	1	0.77
Hadoop DataIO size	1	0.64
Num. of processes	1	0.64
Free swap size	1	1
Cached memory size	0.81	0.77
Buffer memory size	0.65	0.72

Figure 12: The six validated features after the removal of false positives.

In Figure 11(b), both intervals are assigned a "Reference" label. The left one is "Reference" because its frequency is significantly different from the annotated one (3.7 vs. 50.1); while the right one is "Reference" because both its frequency and value difference are quite small, meaning it is similar to the annotated "Reference" interval.

Feature validation. The process of partition discovering and automatic labeling generates a lot more labeled data that helps EXstream filter out false positives, and improve the current set of features. Features that have high entropy reward on the annotated partition will be reevaluated on the large dataset. If the high reward is validated in the larger dataset as well, the feature is maintained; otherwise, it is discarded. In our running example, after the validation step, only 6 out of 670 features remain. Figure 12 shows the reward for each of these 6 features for the annotated partition and the augmented partition set.

5.3 Step 3: filtering by correlation clustering

After the validation step, we are usually left with a small set of features, which have high individual rewards, and the high rewards are likely related to the investigated anomaly. However, it is still possible that several of these features have information overlap. For example, two identical features, are good individually, but putting them together in an explanation does not increase the information content. We identify and remove correlated features using clustering.

We use pairwise correlation to identify similar features. We represent a feature as a node; two nodes are connected, if the pairwise correlation of the two features exceeds a threshold. We treat each connected component in this graph as a cluster, and select only one representative feature from each cluster. In our running example, the final six features are clustered into two clusters, one cluster with a single node, and another cluster with five nodes. Based on this result, the final explanation has two features.

5.4 Building final explanations

Once we make the final selection of features, the construction of an explanation is straightforward. For each selected feature, we can build a partial explanation in the format defined in Section 2.3. The feature name becomes the variable name. The value boundaries for the abnormal intervals become the constants. If a feature offers perfect separation during segmentation (Section 4), there is one boundary and only one predicate is built: e.g., the abnormal value range of feature f_1 is $(-\infty, 10]$, then the predicate is $f_1 \leq 10$. If a feature has more than one abnormal intervals, then multiple predicates are built to compose the explanation: e.g., the abnormal value ranges of feature f_2 are $(-\infty, 20]$,$[30, 50]$, and then the explanations are $f_2 \leq 20 \vee (f_2 \geq 30 \wedge f_2 \leq 50)$. Then

No.	Anomaly	Hadoop workload
1	High memory	WC-frequent users
2	High memory	WC-sessions
3	Busy Disk	WC-frequent users
4	High High CPU	WC-frequent users
5	High High CPU	WC-sessions
6	Busy High CPU	Twitter trigram
7	High Busy Network	WC-sessions
8	High Busy Network	Twitter trigram

Figure 13: Workloads for evaluating the explanations returned by EXstream.

we simply connect the partial explanations constructed from different features using conjunction and write the final formula into the conjunctive normal form.

6. EVALUATION

We have implemented EXstream on top of the SASE stream engine [3, 26]. Due to the space constraints, the implementation details are left to our technical report [27]. In this section, we evaluate EXstream on the conciseness, consistency, and prediction power of its returned explanations, and compare its performance with a range of alternative techniques in the literature. We further evaluate the efficiency of EXstream when the explanation module is run concurrently with monitoring queries in an event stream system.

6.1 Experimental Setup

In our first use case, we monitored a Hadoop cluster of 30 nodes which was used intensively for experiments at the University of Massachusetts Amherst. To evaluate EXstream for explaining anomalous observations, we used three Hadoop jobs: (A) Twitter Trigram: count trigrams in a twitter stream; (B) WC-Frequent users: find frequent users in a click stream; (C) WC-session: sessionization over a click stream.

The running example throughout this paper, which starts to show in Figure 1(a) and 1(b), is a real use case. A Hadoop expert found out the root causes by manually checking a large volume of logs. The expert also confirmed that the results generated by EXstream match the ground truth perfectly.

To enable the ground truth for evaluation further, we manually created four types of anomalies by running additional programs to interfere with resource consumption: (1) High memory usage: the additional programs use up memory. (2) High CPU: the additional programs keep CPU busy. (3) Busy disk: the programs keep writing to disk. (4) Busy network: the programs keep transmitting data between nodes. By combining the anomaly types and Hadoop jobs, we create 8 workloads listed in Figure 13. The ground truth features are verified by a Hadoop expert.

Our second use case is supply chain management of an aerospace company. Due to confidentiality issues we were unable to get real data. Instead, we consulted an expert and built a simulator to generate manufacturing data and anomalies such as faulty sensors and subpar material. Since both use cases generate similar results, we report results using the first use case and refer the reader to [27] for results of the second use case.

All of our experiments were run on a server with two Intel Xeon 2.67GHz, 6-core CPUs and 16GB memory. EXstream is implemented in Java and runs on Java HotSpot 64-bit server VM 1.7 with the maximum heap size set to 8GB.

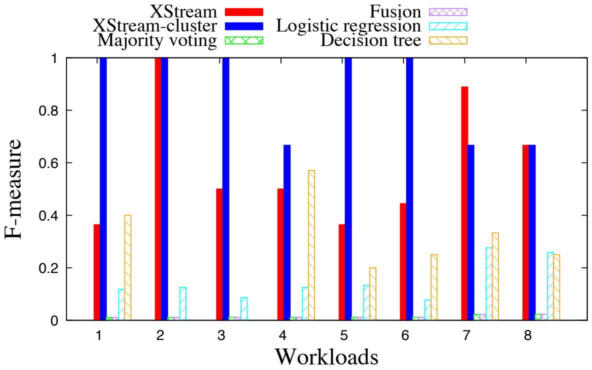


Figure 14: Consistency comparison

6.2 Effectiveness of Explanations by EXstream

We compare EXstream with a range of alternative techniques. We use **decision trees** to build explanations based on the latest version of weka, and **logistic regression** based on a popular R package. We consider two additional techniques, **majority voting** [15] and **data fusion** [19]. Both techniques make full use of every feature, and make prediction based on all features. Majority voting treats features equally and uses the label which counts the most as the prediction result. The fusion method fuses the prediction result from each feature based on their precision, recall and correlations. We compare these techniques on three measures: (1) consistency: selected features as compared against ground truth; (2) conciseness: the number of selected features; (3) prediction accuracy when the explanation is used as a prediction model on new test data.

Consistency. First we compare the selected features of each algorithm with the ground truth features. The results are shown in Figure 14. X-axis represents different workloads (1 - 8), while Y-axis is the F-measure, namely, the harmonic mean of precision and recall regarding the inclusion of ground truth features in the returned explanations. EXstream represents our results before applying clustering on selected features, while EXstream-cluster represents results clustered by correlations (Section 5). We can see that EXstream-cluster works better than EXstream without clustering for most of workloads, and EXstream-cluster provides much better quality than the alternative techniques. Majority voting and fusion do not select features, and hence their F-measures are low. Logistic regression and decision tree generate models with selected features, with sightly increased F-measures but still significantly below those of EXstream-cluster.

Conciseness. Figure 15 shows the sizes of explanations from each solution. Here the Y-axis (in logarithmic scale) is the number of features selected by each solution, where the total number of available features is 345. "Ground truth" represents the number of features in ground truth, while "Ground truth cluster" represents the number of clusters after we apply clustering on the contained features. Again, majority voting and fusion do not select features, so the size is the same as the size of feature space. The models of logistic regression includes 20 - 30 features, which is roughly 10 times of the ground truth. Decision trees are more concise with less than 10 features selected. Overall, EXstream outperforms other algorithms, and is quite close to the number of features in ground truth cluster.

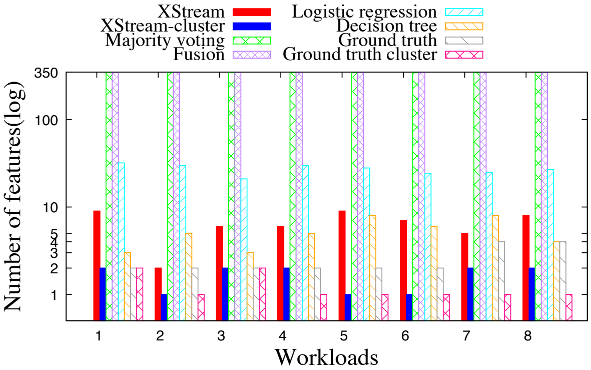


Figure 15: Conciseness comparison

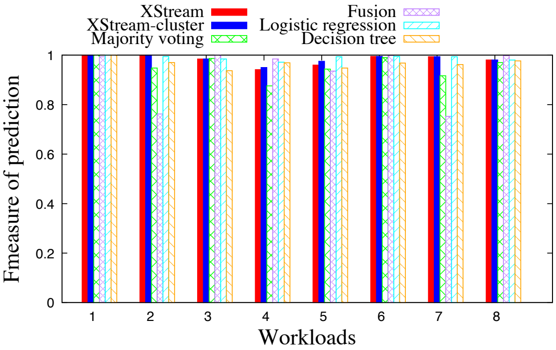


Figure 16: Prediction power comparison

Predication accuracy. In Figure 16 we compare the prediction accuracy of each method. The Y-axis represents F-measure for prediction over new test data. The F-measures of EXstream, logistic regression and decision tree are quite stable, most of time above 0.95. Data fusion and majority voting fluctuate more. Overall, our method can provide consistent high-quality prediction power.

Effectiveness of the distance function. We finally demonstrate the effectiveness of our entropy-based distance function by comparing it with a set of existing distance functions [24] for time series: (1) Manhattan distance, (2) Euclidean distance, (3) DTW, (4) EDR, (5) ERP and (6) LCSS.

The results are shown in Figure 17. In each method, all available features are sorted by the distance function of choice in decreasing order. We measure the number of features retrieved from each sorted list in decreasing order in order to cover all the features in the ground truth, shown as the Y-axis. We see that our entropy distance is always the one using the minimum number of features to cover the ground truth. LCSS works well in the first two workloads, but it works poorly for workloads 3, 4, 5, and 6. This is because the ground truth features for the first two workloads have perfect separating power based on LCSS distance, while in other workloads they contain some noisy signals. So LCSS is not as robust as our distance function. Other distance functions always use large number of features.

Summary. Our explanation algorithm outperforms other techniques in consistency and conciseness while achieving comparable, high predication accuracy. Specifically, EXstream improves consistency to other methods from 10.7% to 87.5% on average, and up to 100% in some cases. EXstream is also more concise, reducing the number of features in an explanation 90.5% on average, up to 99.5% in some cases. EXstream is as good as other techniques on prediction quality: its F-measure on prediction is only slightly worse than logistic regression by 0.4%, while it is 3.3% higher than majority voting, 6.1% percent higher than fusion, and 1.9% higher than decision tree.

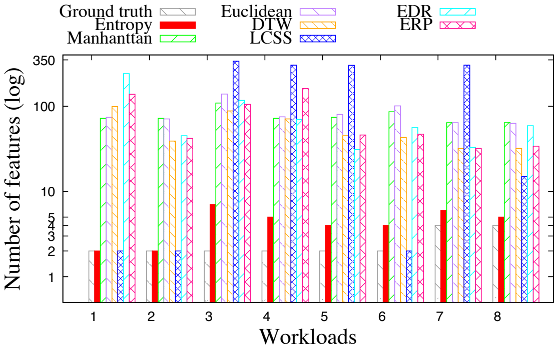


Figure 17: Distance function comparison

Our entropy distance function works better than existing distance functions on time series. It reduces the size of explanations by 94.6% on average, up to 97.2%, compared to other functions.

6.3 Efficiency of EXstream

We further evaluate the efficiency of EXstream. Our main result shows that our implementation is highly efficient: with 2000 concurrent monitoring queries, triggered explanation analysis returns explanations within half a minute and affects the performance only slightly, delaying events processing by only 0.4 second on average. Additional details are available at [27].

7. RELATED WORK

In the previous section, we compared our entropy distance with a set of state-of-the-art distance functions [24] and compared our techniques with prediction techniques including decision trees and logistic regression [2]. In this section we survey broadly related work.

CEP systems. There are a number of CEP systems in the research community [8, 17, 1, 21, 23]. These systems focus on passive monitoring using CEP queries by providing either more powerful query languages or better evaluation performance. Existing CEP techniques do not produce explanations for anomalous observations.

Explaining outliers in SQL query results. Scorpion [25] explains outliers in group-by aggregate queries. Users annotate outliers on the results of group-by queries, and then scorpion searches for predicates that remove these outliers while minimally affect the normal answers. It does not suit our problem because it works only for group-by aggregation queries and it searches through various subsets of the tuples that were used to compute the query answers. As shown for our example, Q1, the explanation of memory usage contention among different jobs cannot be generated from only those events that produced the monitoring results of Q1. Recent work [20] extends Scropion by supporting richer and insightful explanations by pre-computation and thus enables interactive explanation discovery. This work assumes a set of explanation templates given by the user and requires precomputation in a given database. Neither of the assumptions fits our problem setting.

Explaining outputs in iterative analytics. Recent work [7] focuses on tracking, maintaining, and querying lineage and "how" provenance in the context of arbitrary iterative data flows. It aims to create a set of recursively defined rules that determine which records in a data-parallel computation inputs, intermediate records, and outputs require explanation. It allows one to identify when (i.e., the points in the computation) and how a data collection changes, and provides explanations for only these few changes.

Set-based distance function for time series. Besides the lock-step and elastic distance functions we compared with, time series are also transformed into sets [18] for measurement. However, the goal of the set-based function is to speed up the computation of existing elastic distance, so it is different from our entropy based distance function.

Anomaly detection. Common anomaly detection techniques [5, 6, 14, 13, 22] do not fit our problem setting. There are two main approaches. One is using a prediction model, which is learned on labeled or unlabeled data. Then incoming data is compared against with expected value by the model. If the difference is significant, the point or time series will be reported as outlier. The other approach is using distance functions, and outliers are those points or time series far from normal values. Both approaches report only outliers, but not the reasons (explanations) why they occur.

8. CONCLUSIONS

In this paper, we present EXstream, a system that provides high-quality explanations for anomalous behaviors that users annotate on CEP-based monitoring results. Formulated as a submodular optimization problem, which is hard to solve, we provide a new approach that integrates a new entropy-based distance function and effective feature ranking and filtering methods. Evaluation results show that EXstream outperforms existing techniques significantly in conciseness and consistency, while achieving comparable high prediction power and retaining a highly efficient implementation of a data stream system.

To enable proactive monitoring in CEP systems, our future work will address temporal correlation in discovering explanations, automatic recognition and explanation of anomalous behaviors, and exploration of richer feature space to enable complex explanations.

Acknowledgements. This work was supported in part by the National Science Foundation under grants IIS-1421322, IIS-1453543, and IIS-1218524.

9. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] C. C. Aggarwal. *Data Mining: The Textbook*. Springer Publishing Company, Incorporated, 2015.

[3] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, New York, NY, USA, 2008. ACM.

[4] R. S. Barga, J. Goldstein, M. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. *arXiv preprint cs/0612115*, 2006.

[5] L. Cao, J. Wang, and E. A. Rundensteiner. Sharing-aware outlier analytics over high-volume data streams. In *Proceedings of the 2016 International Conference on Management of Data*, pages 527–540. ACM, 2016.

[6] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):15, 2009.

[7] Z. Chothia, J. Liagouris, F. McSherry, and T. Roscoe. Explaining outputs in modern data analytics. *Proceedings of the VLDB Endowment*, 9(4), 2015.

[8] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[9] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. *Fast subsequence matching in time-series databases*, volume 23. ACM, 1994.

[10] U. Fayyad and K. B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. 1993.

[11] U. Feige, V. S. Mirrokni, and J. Vondrak. Maximizing non-monotone submodular functions. *SIAM Journal on Computing*, 40(4):1133–1153, 2011.

[12] Ganglia monitoring system. http://ganglia.sourceforge.net/.

[13] M. Gupta, J. Gao, C. Aggarwal, and J. Han. Outlier detection for temporal data. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 5(1):1–129, 2014.

[14] H. Huang and S. P. Kasiviswanathan. Streaming anomaly detection using randomized matrix sketching. *Proceedings of the VLDB Endowment*, 9(3):192–203, 2015.

[15] L. Lam and S. Suen. Application of majority voting to pattern recognition: an analysis of its behavior and performance. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 27(5):553–568, 1997.

[16] D. Luckham. *Event Processing for Business: Organizing the Real-Time Enterprise*. Wiley, 2011.

[17] Y. Mei and S. Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD Conference*, pages 193–206, 2009.

[18] J. Peng, H. Wang, J. Li, and H. Gao. Set-based similarity search for time series. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2039–2052. ACM, 2016.

[19] R. Pochampally, A. Das Sarma, X. L. Dong, A. Meliou, and D. Srivastava. Fusing data with correlations. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 433–444. ACM, 2014.

[20] S. Roy, L. Orr, and D. Suciu. Explaining query answers with explanation-ready databases. *Proceedings of the VLDB Endowment*, 9(4):348–359, 2015.

[21] StreamSQL Team. StreamSQL: a data stream language extending SQL. http://blogs.streamsql.org/.

[22] L. Tran, L. Fan, and C. Shahabi. Distance based outlier detection for data streams. *Proceedings of the VLDB Endowment*, 9(4):1089–1100, 2015.

[23] D. Wang, E. A. Rundensteiner, and R. T. Ellison. Active complex event processing over event streams. *PVLDB*, 4(10):634–645, 2011.

[24] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, and E. Keogh. Experimental comparison of representation methods and distance measures for time series data. *Data Mining and Knowledge Discovery*, 26(2):275–309, 2013.

[25] E. Wu and S. Madden. Scorpion: explaining away outliers in aggregate queries. *Proceedings of the VLDB Endowment*, 6(8):553–564, 2013.

[26] H. Zhang, Y. Diao, and N. Immerman. On complexity and optimization of expensive queries in complex event processing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 217–228. ACM, 2014.

[27] H. Zhang, Y. Diao, and A. Meliou. Exstream: Explaining anomalies in event stream monitoring tech report. https://cs.umass.edu/%7Ehaopeng/tr.pdf).

Real Time Contextual Summarization of Highly Dynamic Data Streams

Manoj K Agarwal
Microsoft Bing
Search Technology Center - India
Hyderabad – 500032, India
agarwalm@microsoft.com

Krithi Ramamritham
Dept. of Computer Science and Engineering
IIT – Bombay, India
Mumbai – 400076, India
krithi@cse.iitb.ac.in

ABSTRACT

Microblogging streams typically contain information pertaining to emerging real world events. Due to the rapid pace of messages in these data streams, short message size and many concurrent events, it is often difficult for users to understand the full context behind an arriving message. Hence, users resort to the cumbersome task of sifting through many messages to obtain the full context of the underlying event. To address this problem, we propose a novel notion – Contextual Event Summary Threads – and present a technique to extract highly meaningful yet compact event summary threads, capturing the complete context of events appearing in data stream, in real time. Our technique is unsupervised and automatically identifies different facets of live events in an unfiltered data stream in a scalable way and presents them to the users as evolving event threads. Extensive experiments over real data demonstrate that our technique -- while avoiding per message processing -- can summarize live data streams with high accuracy and produce compact event summary threads. The summary size of each event is dependent only on the underlying information and not on the number of messages pertaining to that event. Our technique is generic and is applicable on any chronologically ordered data stream which can be modeled in a <user: message> framework.

CCS Concepts

- **Information systems → Information retrieval → Retrieval tasks and goals → Summarization**

Keywords

Real Time Search; Data Streams; Event Summarization; Algorithm; Experiments.

1. INTRODUCTION

1.1 Motivation

Unstructured data streams -- sequences of chronologically ordered messages posted by multiple users -- occur in various social media and enterprise domains. For example, on Twitter, with a large user base, messages are posted at a high rate. Twitter is often the first medium to report emerging events [14][18]. An event in a data stream is defined by "*messages, posted by multiple users, in the same context, within a bounded time window*", for example, messages posted by the fans during the course of a football match.

An event can be a real world or an abstract activity, relevant for a group of people. It is only natural that in a fast-moving world, a huge number of events occur concurrently.

There have been many recent attempts [14][15][16] at identifying emerging events in real time over live social media streams. Existing unsupervised approaches identify emerging events as temporally and spatially correlated clusters of keywords over dynamic message streams. The temporally and spatially correlated cluster of keywords forms an 'event-topic'. In order to capture the event-topic, a dynamic graph is constructed using the most recent messages, with a sliding window model. Therefore, nodes in the dynamic graph represent temporally correlated keywords. An edge between two nodes -- representing two keywords -- indicates that messages within the recent sliding window contain both the keywords, representing spatial correlation. Thus, nodes of dense sub-graphs embedded in the dynamic graph represent the keyword clusters with strong spatial and temporal correlations [15][16].

When the tweets posted during the Nairobi terrorist attack [30] are fed to the system described in [15], one of the keyword clusters, i.e., event-topic, discovered contained the keywords:

- **A: UK, #kenya, #westgate, #nairobi"**

Clearly, the keywords are insufficient to describe the underlying event. The same is true of another event-topic:

- **B: was, 69, kofi, among, #ghana, attacks, ghanaian, awoonor, killed, poet, prof., #kenya**

Systems such as [14][15] discover event-topics like **A** and **B**. To better understand an event-topic, users have to search for the relevant messages in the data stream by themselves. This burdens the users with the task of understanding the emerging events manually or to determine if there is a connection between **A** and **B**. The shortcomings of the message search-based approach are:

(1) Keyword search over live data streams is primitive, e.g., Twitter just returns the most recent tweets for a given search query [19]. It is not necessary that the most recent tweets are also the most relevant and informative tweets for the event.

(2) Keyword search can produce an information overload for a fast-moving data stream. Often a large number of tweets are returned by Twitter for a search query [5]. Moreover, search results are continuously updated with recent messages. This poses difficulties for the users to keep pace with the evolving events.

Typically, the rate at which messages are generated is high. Hence, even if a subset of the messages in the query response is informative, it is challenging to identify these messages in real-time. The high rate of message arrival, and the fact that the messages are short often makes it difficult for the users to understand the context of a standalone message. Thus, the first goal of this paper is to discover a minimal set of most informative messages from the message stream, related to an emerging event. These messages represent the complete summary of the event. Furthermore, live real-world events are not just point events – they evolve. Our second goal is, for each discovered event, its summary must be updated every time there is any significant change in the event. Note that the events evolving in real time may comprise several different aspects or facets. When these changes in the event summary are arranged temporally, an event thread results, capturing the complete event context with the passage of time.

Given these needs, we present a novel method to automatically extract the Contextual Event Summary Threads in real-time for events unraveling in an unfiltered and fast moving data stream.

In [31], we presented our methodology to create and update an index over discovered 'Contextual Event Summary Threads' in a highly dynamic data stream in real time and enabled keyword search over these events using it. In this paper, we present our technique to summarize the events and to discover the Contextual Event Summary Threads.

1.2 Contributions

Most real-world events comprise several facets. Therefore, the summary for an event is represented as a Directed Acyclic Graph (DAG) that reveals the way the event has evolved. Each node in an event thread, called sub-event, has an associated event-topic similar to **A** and **B**. An event-topic is summarized using a minimal set of most relevant messages discovered from the data stream.

The event thread in Figure 1 was produced automatically by our system [31] from the tweets sent during '**Nairobi Terrorist Attack' [30]**. There are 13 sub-events in the thread in Figure 1. The summary started with a tweet about a mall being attacked, followed by tweets about action against attackers, rumors, claims and counter claims by authorities and citizens, etc., which were discovered in real time. The event thread describes a meaningful chronological sequence of the event. Figure 1 depicts a part of the event thread discovered over 164K tweets. For each sub-event in the event thread, our method identified an appropriate summary.

We identify most appropriate message(s) in the data stream, describing the event-topic. In Figure 1, sub-event 9 corresponds to **A** and sub-event 5 corresponds to **B**. Note, all the keywords for both the event-topics are present in their summaries.

In summary, our system:

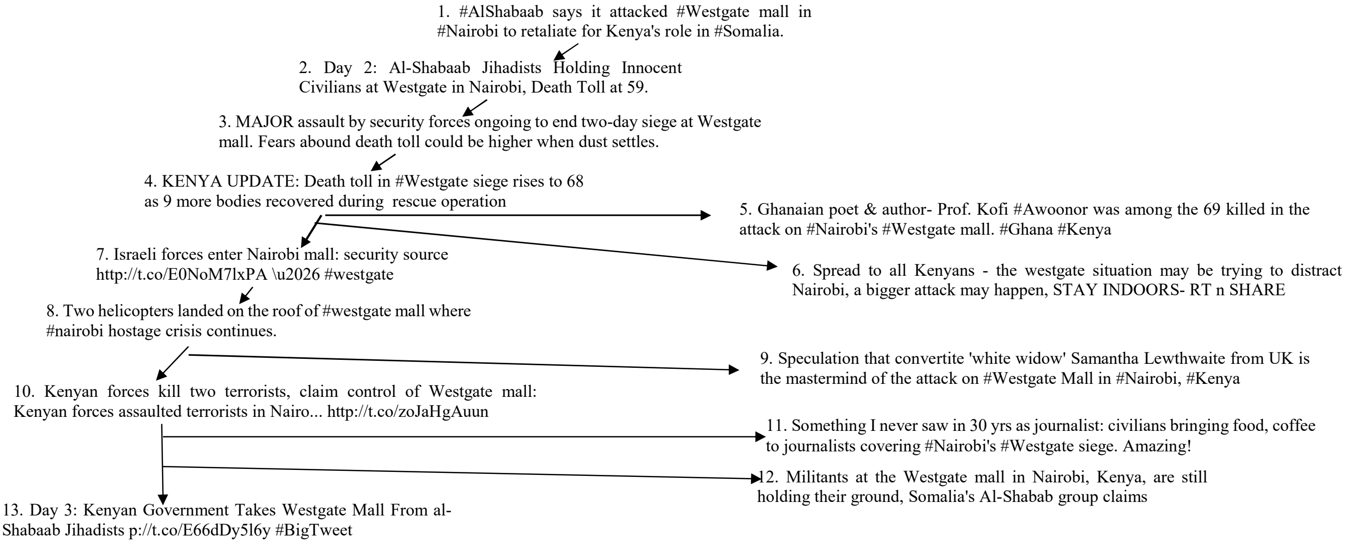
(i) Clusters the related messages together in the data stream: Event-topics are identified by discovering dense sub-graphs, called event-graphs, in the dynamic graph constructed over the message stream [15]. Our system exploits the event-graphs to pool the relevant messages together, related to event-topics.

(ii) Identifies important messages in the message pool to create an event summary: We identify a minimal set of the most relevant messages from the message pool of an event-topic such that the summary is complete and meaningful. In fact, we exploit the structural properties of the underlying event-graph to identify a subset of messages as 'summary candidates'.

(iii) Discovers the event threads for evolving events: As the events evolve, the underlying event-graphs go through structural changes. These changes are tracked in real time. Event summary is updated whenever its event-graph goes through a 'significant' change (cf. Section 6). The updates in the event summaries are captured in contextual summary threads like one shown in Figure 1. Our technique automatically discovers different facets of live events in real time. In general, an event thread can have multiple roots. We say that each unique path in the event thread, from each of its root(s) to each of its leaves, represents a different facet of the event. Event thread in Figure 1 has 6 facets.

In a nutshell, our research contributions involve

1. Summarizing all the events, discovered in a fast-moving unfiltered data stream, in real time, in a scalable and unsupervised manner. For each discovered event-topic, a minimal set of messages is identified to produce a meaningful, informative, stable and complete event summary (Section 5).
2. Tracking the evolution, emergence and dissolution of dense graphs (event-graphs) in a large and highly dynamic graph in the presence of node and edge insertion and deletion.
3. Exposing different facets of live events which are presented as a contextual event summary thread, representing one or more aspects of the event in real time (cf. Section 6).



1. #AlShabaab says it attacked #Westgate mall in #Nairobi to retaliate for Kenya's role in #Somalia.

2. Day 2: Al-Shabaab Jihadists Holding Innocent Civilians at Westgate in Nairobi, Death Toll at 59.

3. MAJOR assault by security forces ongoing to end two-day siege at Westgate mall. Fears abound death toll could be higher when dust settles.

4. KENYA UPDATE: Death toll in #Westgate siege rises to 68 as 9 more bodies recovered during rescue operation

5. Ghanaian poet & author- Prof. Kofi #Awoonor was among the 69 killed in the attack on #Nairobi's #Westgate mall. #Ghana #Kenya

6. Spread to all Kenyans - the westgate situation may be trying to distract Nairobi, a bigger attack may happen, STAY INDOORS- RT n SHARE

7. Israeli forces enter Nairobi mall: security source http://t.co/E0NoM7lxPA \u2026 #westgate

8. Two helicopters landed on the roof of #westgate mall where #nairobi hostage crisis continues.

9. Speculation that convertite 'white widow' Samantha Lewthwaite from UK is the mastermind of the attack on #Westgate Mall in #Nairobi, #Kenya

10. Kenyan forces kill two terrorists, claim control of Westgate mall: Kenyan forces assaulted terrorists in Nairo... http://t.co/zoJaHgAuun

11. Something I never saw in 30 yrs as journalist: civilians bringing food, coffee to journalists covering #Nairobi's #Westgate siege. Amazing!

12. Militants at the Westgate mall in Nairobi, Kenya, are still holding their ground, Somalia's Al-Shabab group claims

13. Day 3: Kenyan Government Takes Westgate Mall From al-Shabaab Jihadists p://t.co/E66dDy5l6y #BigTweet

Figure 1: Contextual Event Summary Thread Discovered by our system for Nairobi Attack

To the best of our knowledge, ours is the first technique that captures the multi-faceted summary of live events and to arrange them in event threads. The event topics, discovered over an unfiltered and fast moving data stream, are summarized in real time in an unsupervised manner, in absence of any user query. The contextual event summary thread is a novel concept in contrast with the story-line generating techniques [4]. Our approach is generic and applicable on any data stream that is a sequence of <user: message>s. Experiments over real data show, our technique leads to compact and meaningful event summary threads for live events.

The organization of the paper is: In Section 2, we present the related work. The event-graph model to capture the state of dynamic data stream and the challenges involved in discovering event summaries are presented in Section 3 and Section 4 respectively. In Section 5 and Section 6, we present the methodology to discover event summary and event summary threads. We present our experimental study in Section 7 followed by conclusion in Section 8.

2. RELATED WORK

Topic Detection and Tracking [1][2] is related to our work. In [12], Yang et al., presented a method to summarize web documents. For these techniques, the document characteristics are completely different from microblog streams (‘large size/less in number/offline’ vs. ‘small size/too many of them/real time’). These methods work on static data only.

In [5], Shou et al., presented a technique to summarize a Twitter data stream, filtered in the context of a given user query. In [23], authors reduce the summarization problem to that of optimizing probabilistic coverage on static data. In [8], Yang et al., present a method to compress the Twitter data stream. Lee et al., in [24] propose a snapshot based method to track the incremental evolution of event-topics. There is significant work [10][11][13] to identify a fixed size summary of microblog posts on a pre-specified topic and on static data, i.e., data which is not updated with new messages.

Gao et al., [3] proposed an unsupervised approach to summarize events, by preparing a joint sentence-tweet level model, across news media and Twitter. Too similar or too different sentence/tweet pair were discarded. In [4], Lin et al. proposed a mechanism to generate a storyline from a microblog data stream. However, the technique is applicable only for static data sets. Vosecky et al., [7] presented a method to identify multiple facets of an event, i.e., important keywords, entities, location, etc., on a static Twitter stream.

The basic difference between existing work and the problem addressed in this paper is: These techniques [1][4][5][7] work in the context of a given query, topic and/or on static data. In contrast, our technique organizes the event summary into event threads, capturing the complete context, for each underlying event discovered in the unfiltered data stream in real time.

The dynamic graph model is exploited to identify the event summary. Our technique exploits the presence of Short Cycle Property (SCP) in the dense sub-graphs of a dynamic graph. In [15], it was shown that dense sub-graphs of a graph possess SCP. There is a large body of work for triangle counting and triangle listing in graphs. Triangle listing is considered an important measure for discovering dense neighborhood [28]. A triangle in a graph induces a cycle of length three. SCP extends this notion to cycle of up to length four, i.e., each node in a graph, possessing short cycle property, participates in a cycle of length at most four within the graph. Triangle listing problem can be divided into processing static graphs [20] and streaming graphs [27]. Streaming graph algorithms for triangle listing are further divided into edge insert graphs [27] and edge insert or delete graphs [9]. In contrast, our algorithm works on a graph with insertion and deletion of both edges and nodes. In this paper, we identify dense event-graphs, discovered based on SCP [15], representing emerging events in a large dynamic graph, with edge and node insert and delete. Our technique keeps track of emergence, evolution and dissolution of the dense sub-graphs in a large dynamic graph. This evolution is presented as contextual event summary thread for the event.

3. EVENT-GRAPH MODEL

Let $S = S_{t\text{-}w}S_{t\text{-}w+1}\ldots S_t$ represents a message stream. $S_i = \{d^i{}_1 d^i{}_2 \ldots d^i{}_m\}$ is the set of messages arriving in block *i.* A message $d^t{}_j$=<u_i, ***k***> in the data stream contains message ***k*** and the *userid* u_j of the user posting the message. A message ***k*** is an ordered set of keywords. *m*, called the *block size*, is the number of messages in a block. The stream is moved forward by expiring the messages in (*t-w*)*th* block and including the messages in (*t*+1)*th* block. The message timestamp is implicit with the arrival order in the data stream. For constructing the event thread as shown in Figure 1, we extract the <userid: message> from the incoming data stream. No pre-processing is done on the message, except removing stop-words from it and tokenizing the message into keywords.

Temporal **and** ***Spatial*** **correlation**: The data stream *S* is modeled as dynamic graph G^t (V^t, E^t) [15]. For the problem studied in this paper, G^t is an input to our system. The graph G^t (V^t, E^t) captures the state of the data stream at the arrival of messages in block *t* ($t \in N^+$). Keywords are represented as nodes in the graph G^t. At time step *t*-1, the graph is updated with the keywords present in the last block of *m* messages and *t*-1 is incremented to *t*. V^t *contains* the *bursty* and *active* keywords in the last *w* blocks at time *t.* Nodes V^t capture *temporal* correlation as only temporally correlated keywords (nodes) are present in the graph.

Table 1: Notation

S_t;$\lvert S_t\rvert$=m	S_t is the block of *m* messages in the t^{th} time step.
G^t=(V^t, E^t)	Dynamic graph at time step *t.* V^t represent the keywords and edges E^t the keyword-correlation in the graph G^t.
$G^t{}_c$($V^t{}_c$, $E^t{}_c$)	$G^t{}_c$ is the event-graph embedded in G^t for event *c* at time step *t*. $V^t{}_c$ represent the keywords in event-topic.
N (*v*); $v \in V^t{}_c$	*N*(*v*) represent the set of adjacent nodes to node $v \in V^t{}_c$.
$d^t{}_j$=<u_i, ***k***>; $1 \le j \le m$	Message $d^t{}_j$ posted by user u_i during time step *t.* ***k*** is set of keywords in the message.
w	Graph G^t contains the bursty and active keywords from last *w* message blocks.
γ	Keyword Burtstiness threshold.
λ	Edge correlation threshold.
$M^t{}_c$	Message set representing summary of event *c* at time *t.*
U_v	Set of userids associated with node $v \in V^t$.
userCover	Set of userids, whose messages include *all* the keywords in an event-topic.

A keyword is *bursty* if it is used in γ (>1) messages in the current window of *m* messages. A keyword *k* is called *active* if its corresponding node $v_k \in V^t$ is present in G^t (V^t, E^t), i.e., the keyword is used in at least one of the messages in the last *w* message blocks. An active keyword remains in the graph G^t for as long as it is active. A keyword is removed from the graph G^t if it is inactive for *w* time windows. Hence, a keyword has to be *bursty* at least once to move into the graph G^t. The burstiness constraint helps identify events relevant for a community or group of users.

Each node $v_k \in V^t$ is associated with a user list U_k, containing *userids* of the users who have used the corresponding keyword k since the time it has moved into the graph G^t. This list is used to establish *spatial correlation* among the keywords (nodes). For a pair of nodes $\{v_i, v_j\} \in V^t$, if the similarity score between their respective user lists U_i and U_j is above a given threshold λ, an edge e is placed between the nodes. *Jaccard coefficient (J_c)* is used as the similarity measure. J_c between two nodes v_i, v_j in graph G^t is calculated as $|U_i \cap U_j| / |U_i \cup U_j|$; hence $0 \leq \lambda \leq 1$. Edge $e \in E^t$ captures the *spatial* correlation between the associated keywords.

Dense sub-graphs in graph G^t represent an event: Dense sub-graphs, embedded in the graph $G^t (V^t, E^t)$ are called event-graphs. The nodes $V^t_c \subset V^t$ in the event- graph $G^t_c (V^t_c, E^t_c)$ are the keywords in the event-topic c and edges $E^t_c \subset E^t$ represent the correlation between these keywords. G^t_c represents the keywords with strong spatial and temporal correlation. In a data stream, modeled as dynamic graph, emerging dense sub-graphs represent the emerging events [15][16].

Short cycles in dense graphs: The length of the Shortest-cycle in a graph has a strong correlation with graph density. For example, triangle listing is a well-known method to measure graph density [27]. Each edge of a triangle induces a cycle of length 3. Similarly, in chordal graphs [29], each node is part of a cycle of length 3. In [15], authors expose a property for the dense sub-graphs embedded in a large graph, called the *short-cycle property*; each node in a dense sub-graph participates in at least one *cycle* of length at most 4, within the sub-graph. Next, we formally define the *Shortest-cycle* and *short-cycle property*.

Def. 3.1.1 *Shortest-cycle*: For a graph $G (V, E)$ and a node $v \in V$, *Shortest-cycle* is the *shortest path P* through which one can return to node v; $\forall e \in P, e \in E$.

Def. 3.1.2 *Short Cycle Property* (*SCP*): For a keyword cluster c, let $G^t_c (V^t_c, E^t_c)$ represent the corresponding subgraph embedded in graph G^t (V^t, E^t) $(\forall k \in c; v_k \in V^t_c)$. $V^t_c \subseteq V^t$, $E^t_c \subseteq E^t$ and $\forall e_{(u,v)} \in E^t_c, \{u, v\} \in V^t_c$. G^t_c possesses the *short-cycle property* if it satisfies the following conditions:

P1. For any two adjacent nodes u and v in the G^t_c, there exists at least *one more* path P between u and v, such that $|P| \leq 3$ and $\forall e_{(u,v)} \in P, e_{(u,v)} \in E^t_c$. Therefore, the length of *Shortest-cycle* for $\forall v \in V^t_c$ is ≤ 4. Note, length of the Shortest-cycle for node v is $|P| + 1$ (for edge $e_{(u,\, v)}$).

P2. For a node $v \in V^t_c$ in graph G^t_c, *all the Shortest-cycles* node v participates in within the graph are of length at most 4.

P3. There is no articulation point in G^t_c. An articulation point is a node whose removal breaks the graph into multiple disconnected components (e.g. in Figure 3(a), node a_4).

In Figure 2 (a), in the absence of P3, clusters C_1 to C_4 will merge into a single cluster since the merged cluster satisfies P1 and P2. Similarly, in the absence of P2, C_1 to C_5 will merge together into a single cluster. Please note, due to SCP, though necessary, it is not sufficient for two event-topics to just share two or more keywords to merge into a single event-topic. In Figure 2(b), C_6 and C_5 share two keywords, but they are not merged together since the merged cluster does not satisfy P2 of SCP. Thus, SCP ensures discovery of dense clusters *efficiently* [15].

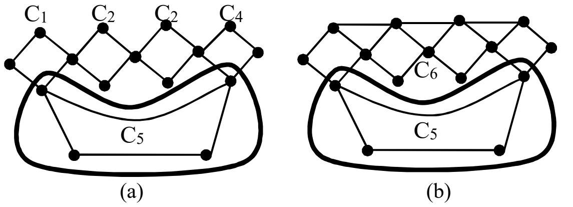


Figure 2: Example event-graphs

Example 1: In Figure 3(a), sets $\{a_1, a_3, a_4\}$ and $\{a_2, a_4, a_5\}$ represent an independent event each. Event-graph in Figure 3(b) represents a single event. In Figure 3(b), keywords having their $J_c \geq 0.25$, have an edge between them. An event-graph satisfies the *SCP* (Def. 3.1.2).

In Figure 3 (b), k_2 and k_3 participate in two *Shortest-cycles* each. Cycle $k_1 \rightarrow k_2 \rightarrow k_4 \rightarrow k_3 \rightarrow k_1$ is an intra-graph cycle too but not the *Shortest-cycle* (Def. 3.1.1). In [15], authors present an efficient algorithm to identify dense-graphs in a dynamic graph by exploiting SCP. SCP ensures that keywords that show a strong temporal and spatial correlation are identified as event-topics.

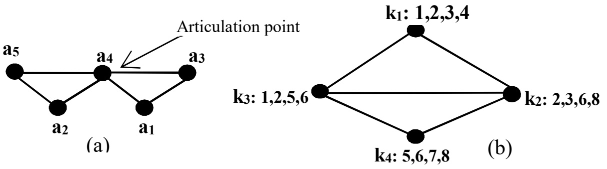


Figure 3: Example event-graphs

4. ISSUES in EVENT SUMMARIZATION

There is a user list U_v associated with each node $v_k \in V^t_c$ in event-graph $G^t_c (V^t_c, E^t_c)$ for event c. v_k is the node corresponding to keyword k. We represent v_k by just v when the context is clear. The messages posted by users in list $U_v, \forall v \in V^t_c$, are considered the pool of message related to event c. Event summary is identified from these messages.

Def. 4.1 Event Summary: Event summary is a *set of messages* $\mathbf{M}^t_c$ from a set of users $U^c \subseteq (\cup U_v; \forall v \in V^t_c)$ such that $\forall v_k \in V^t_c$, $\exists u \in U^c$ where u has used the keyword k in its message(s).

Thus, the event summary is defined as a set of message(s) from a set of users whose messages covers all the keywords in the event-topic. This set of users is also called the valid *userCover*.

Let $u \in U_v$ of the userid list of node $v_k \in V^t_c$. Let $N(v_k)$ represents the set of nodes adjacent to v_k in $G^t_c (V^t_c, E^t_c)$. $\forall n \in N(v_k)$, if $u \in U_n$, node n is considered covered. Thus, all neighbors of node v_k that contain u in their respective user lists are considered covered. Note, *only* the user lists of the neighbors of node v_k in the event-graph are checked for the presence of user u. The messages in the current time window from the users in the *userCover* become part of the event summary $\mathbf{M}^t_c$. The summary $\mathbf{M}^t_c$ for a new event c, emerging in the t^{th} time window is defined as $\mathbf{M}^t_c = \bigcup S^t_u \mid u \in U^c$ where S_u^t is a set of messages from user u in t^{th} message block. $\mathbf{M}^t_c$ is the ordered sequence of messages based on the message timestamp.

Def. 4.3 *Optimal UserCover*: An optimal *userCover T* is the smallest subset of users such that $\forall U_v \mid v \in V^t_c, U_v \cap T \neq \phi$.

Theorem 1: *Discovering optimal userCover is NP-hard.*

Proof Sketch: Hitting set is a well-known NP-complete problem [25]. It is defined as follows: Given a collection *S* of sets $S_i s$; $1 \le i \le n$, find the smallest size set *H* such that $\forall S_i \in S, S_i \cap H \neq \phi$. By mapping each element in a set S_i to a user id in *userCover,* Hitting set problem is polynomial time reducible to *userCover*. If T* is a solution for *userCover* and H* is a solution for Hitting set, |H*| = |T*|. □

Theorem 1 states, discovering optimal *userCover* is NP-hard even when the user lists associated with keywords are static. However, the set of nodes V^t_c in an event graph G^t_c as well as the user lists U_v associated with each node $v \in V_c^t$ are highly dynamic due to continuous updates in the message stream. Thus, it is non-trivial to identify a minimal set of most relevant messages from the data stream that ensures that the event summary be informative, compact, complete, meaningful and stable.

Identifying users (and not messages) helps us create a more meaningful event summary (at times, users post multiple messages in a small time window to explain the full context of their messages). Since user lists are large for popular events, it is non-trivial to identify optimal *UserCover*. To efficiently identify these users, each arriving message is assigned a score in a scalable manner to rank more relevant messages higher (cf. Section 5.3).

We next highlight how the dynamic updates in the data stream may result in an unstable event summary:

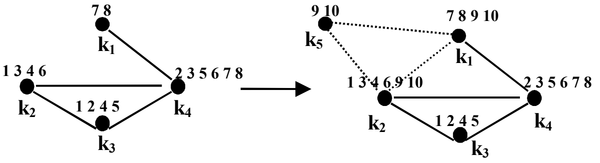


Figure 4: Evolution in the event-graph

Live updates can make a summary unstable: For an event present in a live data stream, let its initial event-graph be as shown in Figure 4(a). The *userCover* had been identified as {1, 3} by any *userCover* discovery algorithm (**k₁** in Figure 4(a) is not part of the event-graph). In the next time step, the event-graph gets updated to as shown in Figure 4(b). However, the same algorithm identifies {2, 9} as *userCover*, which results in a different set of messages as summary, making it difficult for users to keep track of evolving event. Thus, due to live updates, event summary may be unstable.

The evolution of the summary for a live event is handled as described in Section 6. We present our algorithm to summarize the event by identifying an approximate *userCover* in Section 5.

5. DISCOVERING EVENT SUMMARIES

In this section, we present our method to discover meaningful event summaries, with the aid of central nodes, called pivot nodes, in the dense event-graphs. In Section 5.2, we present our algorithm to identify the event summaries. In Section 5.3, we present our method to rank the messages in the data stream.

5.1 Pivot Nodes and Pivot Edges

Def. 5.1.1 ***Pivot Edge***: An edge that participates in more than one *Shortest-cycles* within the event-graph is defined as a pivot edge. Higher the number of Shortest-cycles a pivot edge participates in, the more central it is in the event graph.

Def. 5.1.2 ***Pivot Node***: Nodes associated with a pivot edge are called *pivot nodes.*

Nodes which are not pivot nodes are called *peripheral* nodes. In Figure 4(b), edge (**k₂**, **k₄**) is a pivot edge. {**k₂**, **k₄**} are pivot nodes.

Lemma 1: *A graph possessing short-cycle property with more than one shortest-cycles within the graph, has a pivot edge.*

Proof: Let event-graph G^t_c (V^t_c, E^t_c) has multiple short cycles, satisfying SCP. Let *C* (*V, E*) be a cycle in the event-graph G^t_c such that it has no common edge with any other cycle in the graph. Let $n \in C(V)$ be a node common with another cycle in graph G^t_c (if cycle *C* has not even one node common with any other cycle within the graph; G^t_c will be disconnected); Since no edge in cycle C participates in another cycle, for any node *v* in *C*(*V*) - *n*, and for any node *u* in V^t_c – *V*, there exist just one path from *v* to *u* via node *n*. Hence in graph G^t_c, node *n* is an articulation point, violating Def. 3.1.2 of SCP event-graph. Therefore, there must exist another path from *v* to *u*. Hence, there exist a cycle $v \to n \to u \to v$. Hence any edge $e \in C(E)$ in path $v \to n$ participates in another cycle, i.e., *e* is a pivot edge. Therefore, for each Shortest-cycle *C* (*V, E*) in an event-graph, $\exists e \in C(E)$ that is a pivot edge. □

Corollary of Lemma 1: *Either a node itself or one of its neighbors is the pivot node in the event-graphs that possesses SCP.* Due to this corollary, we can create a pivot edge cover (*PECover*), as described below.

Def. 5.1.3 PECover: For G^t_c (V^t_c, E^t_c), a *PECover* is a subset of pivot edges $E_p^o \subseteq E^t_c$ such that, $\forall v \in V_c^t$, $\exists e(u,v) \in E_c^t \mid v \in V_p^o$.

Hence, a *PECover* (corresponding *PNCover*) E_p^o (V_p^o) is a subset of pivot edges (corresponding pivot nodes) in the event-graph G^t_c (V^t_c, E^t_c) such that $\forall v \in (V_c^t - V_p^o)$, node *v* is adjacent to a node in V_p^o.

Let U_v be the user list for node $v \in V^t_c$ and *C* be a collection of all the user lists in G^t_c (V^t_c, E^t_c). Let C_p be a collection of user lists associated with pivot nodes V_p^o in graph G^t_c ($C_p \subset C$).

Lemma 2: *For collection* C_p $\forall U_i \in C - C_p; \exists U_j \in C_p \mid$, $U_i \cap U_j \neq \phi$.

Proof: For two nodes v_i and v_j and their associated user lists U_i and U_j in an event-graph G^t_c(V^t_c, E^t_c), if edge (v_i, v_j) $\in E^t_c$, $|U_i \cap U_j| \ge \lambda . |U_i \cup U_j|; \lambda > 0, |U_{l \in \{i,j\}}| \ge \gamma$. Therefore, for any edge $e \in E^t_c$ there exists at least one userid which occurs in the user lists of both the nodes. Since, each node in G^t_c is adjacent to a node in the *PECover* of G^t_c, therefore, for collection C_p, $\forall U_i \in C - C_p; \exists U_j \in C_p \mid U_i \cap U_j \neq \phi$. □

Hence, we first identify a set of pivot edges, called *PECover*. A valid *userCover* can be identified *only* from the user lists associated with pivot edges (cf. Lemma 2). Thus, pivot edges help us avoid processing a large number of messages associated with *peripheral* nodes and yet ensure that the event summary is complete.

Further motivation to use the *pivot edges* is explained below:

5.1.1 Informative

A message containing more keywords from the event graph is considered more informative. Naturally, a keyword with higher number of neighbors in the event graph is likely to contain such messages. We capture this notion of informative-ness as follows:

Def. 5.1.4 ***Informative-ness*****:** Informative-ness of a node v is defined as $N(v)$, the set of nodes adjacent to v in event graph $G^t{}_c$.

Let edge *e (u, v)* be a pivot edge. Let s be a neighbor of u such that *e* (*u, s*) be a non-pivot edge (*s* is a *peripheral* node). $N(v)$ represents the set of nodes adjacent to v in G^t *(*V^t*,* E^t*).*

Lemma 3: $|N(u)| > |N(s)|$.

Proof: Omitted (cf. Lemma 5). □

Lemma 3 shows that the messages from the users associated with the pivot nodes contain more keywords from the event-topic. Identification of summary from these messages leads to a more informative and compact summary.

5.1.2 Stable

Since a pivot edge participates in multiple cycles, it is more stable than the other edges in a dynamic event-graph. Even if one or more edges/nodes, among edges and nodes adjacent to a pivot edge get deleted in the underlying event-graph, the message set identified as the event summary continues to be a valid event summary (cf. Section 6.2). Hence, identification of event summary based on pivot edges, leads to more stable event summary.

5.1.3 Complete

An event summary containing messages that cover all the keywords in an event-graph is considered a complete event summary. However, we identify the users only from the user lists associated with pivot nodes, i.e., from the collection C_P.

Lemma 4: *A user cover identified only from the pivot nodes of an event-graph, possessing short cycle property, is a valid userCover.*

Proof: The corollary of Lemma 1 along with Lemma 2 completes the proof. □

Lemma 4 shows that the *userCover* identified from the user lists in collection C_P results in a valid *userCover*. With the help of pivot edges, we identify a subset of user lists C_P (from the entire user list collection C for event c) and a valid *userCover* can be identified only *from* these sets. Thus, the event-graph structure helps us identify a small number of more relevant user lists for discovering a valid *userCover*. However, discovering optimal set of pivot nodes in an event-graph remains an NP-hard problem (cf. Theorem 3).

5.1.4 Optimal Summary Size

We next show that the summary size discovered with the aid of optimal *PECover* leads to the optimal size summary.

As shown in Lemma 2, once a pivot edge $e_{(u,\,v)}$ is identified, all the nodes adjacent to the pivot nodes $\{u, v\}$ can be covered using only the user lists associated with these nodes. We now show that there cannot be any smaller collection of user ids than optimal size *PECover* that results in a valid *userCover*.

Let T* represent the optimal collection of user ids that provide a valid *userCover* for a given event-graph G^t and let *PECover** be the optimal *PECover.*

Theorem 2: $|PECover^*| \leq |T^*|$.

Proof: For a given event-graph G^t (V^t, E^t), a dominating set D is subset of nodes in V^t such that each node in *(*$V^t - D$*)* is adjacent to a node in D [25]. Let $D^* \subset V^t$ be the optimal size dominating set. Therefore, D^* is the smallest set of nodes such that each node in $V^t - D^*$ is adjacent to a node in D^*.

Step 1: Each node in V^t*,* is adjacent to a pivot edge (corollary of Lemma 1). Therefore, for each node v in D^* we get a corresponding pivot edge as follows: either edge $e(v, u)$ is a pivot edge; $u \in N(v)$ or edge e (u, $N(u)$) is a pivot edge.

In this manner, we get a pivot edge cover *peCover* from D^* which is a valid PECover since D^* is a dominating set for graph G^t; $|peCover| = |D^*|$. Let *PECover** is the optimal *PECover.* Hence, $|PECover^*| \leq |peCover| \leq |D^*|$.

Step 2: Any two nodes that share an edge, have at least one userid common (Lemma 2). Let T^* be the optimal set of user ids, providing the valid *userCover*. Since D^* is the optimal dominating set, therefore, $|T^*| \geq |D^*|$ because for each node in D^*, at least one userid has to be selected in T^* (since the user lists of *only* the neighbors of a node in the event-graph are considered while constructing *userCover*)

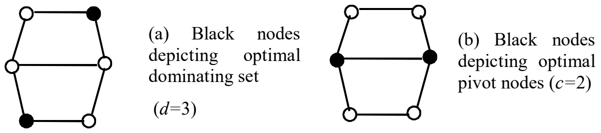
From Step 1 and Step 2, $|PECover^*| \leq |T^*|$ □

Thus, optimal pivot edge cover leads to optimal summary size.

On event graph $G^t{}_c$ we induce a graph $G'(V', E')$ such that each edge in G' is a pivot edge in $G^t{}_c$ (with the aid of Lemma 5, Section 5.2). We identify a smallest subset $V_{PN} \subset V'$ of nodes in G' such that every node in $V' - V_{PN}$ is adjacent to at least one node in V_{PN}. We call the set V_{PN} Pivot nodes cover or *PNCover*.

Theorem 3: *Discovering optimal PNCover for a given event-graph* $G^t{}_c(V^t{}_c, E^t{}_c)$ *is NP-hard.*

Proof Sketch: Dominating set is a NP-complete problem [25]. It can be shown that $Do\min atingSet \leq_P PNCover$. □



(a) Black nodes depicting optimal dominating set

(d=3)

(b) Black nodes depicting optimal pivot nodes (c=2)

Figure 5: Dominating set Vs. Pivot nodes over event-graph

An alternative to pivot edges is to identify the dominating set of nodes [25] itself in an event-graph. Our primary reason to choose the pivot edges over dominating set is, nodes in dominating set divides a graph into stars whereas the pivot edges divide it into cycles (i.e., quasi-cliques). The *userCover* discovered from nodes that are part of same quasi-clique leads to more informative event summary. For instance, black nodes in Figure 5 (a) and Figure 5 (b) both represent the dominating sets. However, Figure 5 (b) represents the dominating set induced due to the pivot edge. Therefore, *pivot edges* ensure that a message summary discovered based on *pivot edges* is more informative, stable, complete, and compact.

Pivot nodes are ‘central nodes’ in an event graph. There are other notions of central nodes in a graph, for example, HITS [17] and PageRank [21] identify central nodes (authorities). However, there are basic differences in the settings of our two problems as:

a) [17][21] techniques are iterative in nature, therefore not amenable to rapidly changing dynamic graphs;

b) The notion of ‘completeness’ (Section 5.1.3) is not applicable for these methods. Due to the same reason, the notion of betweenness centrality [22] is not applicable;

c) These techniques exploit the link structure to identify the authorities, whereas we exploit the graph structure to identify the pivot nodes (since the objectives of two problems are different).

5.2 Approximate User Cover

Next, we present our algorithm to identify approximate *userCover*. On an event-graph $G^t_c(V^t_c, E^t_c)$, we induce a graph $G^{'}(V^{'}, E^{'})$, $V^{'} \subseteq V_c^t$ and $E^{'} = E_c^t \bigcap (V^{'} \times V^{'})$ such that $\forall v \in V^{'}$; v is a pivot node in graph G^t_c. The induced graph $G^{'}$ is called the *core* event-graph. The event summary discovered from $G^{'}(V^{'}, E^{'})$ follows the same notion of event-summary (Def. 4.1) except that it is discovered on induced graph $G^{'}$.

Instead of identifying the *userCover* for graph G^t_c, we identify the *userCover* on *G'*. The core event-graph *G'* does not contain the *peripheral nodes* in G^t_c. However, presence of peripheral nodes in the event-graph G^t_c defines the pivot nodes. Therefore, peripheral nodes impact the event summary *only* indirectly.

Lemma 5: A node *v* with degree $\Delta(v) > 2$ in graph $G^t_c(V^t_c, E^t_c)$ is a pivot node.

Proof: $\forall v \in V_c^t, \Delta(v) \geq 2$ (node *v* is part of a cycle). Let $v \in V_c^t | \Delta(v) > 2$ be a node in graph G^t_c such that *v* is not a pivot node. Since $\Delta(v) > 2$, node *v* is part of more than one cycles. Let C_1 and C_2 be two cycles node *v* is part of. Let edge $e_{(v, n')}$ belong to C_1 and edge $e_{(v, n'')}$ belong to C_2. Hence, there exists one path from node *n'* to node *n''* via node *v*. However, if this is the only path between the two nodes, node *v* becomes an articulation point violating the short cycle property. Hence there exists one more path *p* from node *n'* to *n''* inducing a cycle C $(v \rightarrow n' \xrightarrow{p} n'' \rightarrow v)$. Length of cycle $|C| \leq 4$ (*SCP*). Therefore, edge $e_{(v, n')}$ participates in two cycles C_1 and *C*. Hence $e_{(v, n')}$ is a pivot edge and *v* is a pivot node. □

With the aid of this lemma, it is easy to induce graph *G'* on G^t_c. Note, *G'* may not follow the *short cycle property*.

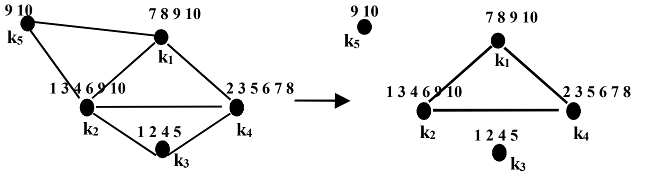


Figure 6: Discovering core graph from an event-graph

Each node *v* in *G'* is assigned a score *p*(*v*); $p(v) \leftarrow d_v(G^t_c)$; $d_v(G^t_c)$ is the degree of node *v* in graph G^t_c. This score is used to identify *userids* in the *userCover*. In Lemma 2, it is shown that if two nodes are adjacent, they have one or more userids common. Therefore, by selecting the nodes with higher score, event summary is likely to contain more keywords in event-graph G^t_c. We describe our algorithm to find *userCover* for graph *G'* below.

Algorithm `approxUserCover (nodeList nL)`

```
1.  Let $G^t_c(V^t_c, E^t_c)$ be an event-graph.
        a.  let cId be its clusterId;
2.  G' is the core event-graph induced on graph $G^t_c$.
        a.  $\forall v \in G^{'}(V^{'}, E^{'}); p_v \leftarrow d_v(G_c^t)$
        b.  $U_v \leftarrow$ list of userids associated with node $v$
3.  $S \leftarrow \Phi$
4.  for $\forall v \in G(V^{'})$ {if $v \notin nL$ { $S \leftarrow S \bigcup v$ }}
5.  cId.nL ← V'; /*we record the pivot node to efficiently update
                       the event summary when cluster evolves later*/
6.  uC ← Φ /*userCover is set to null*/
7.  while ( $S \neq \phi$ )
        a.  $v \leftarrow \arg Max_{v \in V^{'}} (p_v \times d_v(G^{'})$ /*return node with
                                          highest $p_v \times d_v(G')$ score*/
        b.  $U \leftarrow U_v$ ;
        c.  Let $U_n$ be userids set associated with a node n, such that n is
            neighbor of node v in G'.
                i.   T ← $\arg Max_{t \in U \bigcap U_n} messageRank(t)$
                ii.  uC ←uC ⋃ {T};
                iii. $U = U - U \bigcap U_n$ /*We do not select any more userids
                     covering same keyword*/
        d.  For each remaining neighbor, check if userid T exists in their
            userid list
                i.   A← v.adjList (G') /*adjacency list of v in G'*/
                ii.  $\forall u | u \in A - n$, if $T \in U_u$, S ←S-u;
8.  return uC;
```

Each edge in graph *G'* is a pivot edge in G^t_c. The degree of a node *v* in *G'*, $d_v(G')$ is a measure of the number of cycles it participates in graph G^t_c. *messageRank* (.) returns the userid of the message with highest rank. Since, a user is added in the user list of a node one at a time, it is kept sorted in the message score efficiently. In the above algorithm, each node in the event-graph is visited *exactly* once. Thus, the complexity of identifying *userCover is* $O(|V^t_c|)$. Algorithm `approxUserCover` is greedy and achieves the same complexity as Dominating set [25], i.e., (1+ log (Δ)) OPT; OPT is optimal *userCover* size and Δ is the maximal degree in graph *G'*. Dominating set problem is LOG-APX-COMPLETE and no better bound is possible.

5.3 Ranking the Messages

Since our objective is to summarize a highly dynamic data stream in real time, the methodology to rank each arriving message must be i) fast and efficient; ii) rank more meaningful messages higher; and iii) does not re-rank the already ranked messages with fresh updates in the data stream. There are many studies related to ranking the microblogs [18][19][6]. A common conclusion across these studies is that the rank of a tweet depends on the authority of its author and the authority of the message. We exploit these features to establish the rank of a tweet.

Let $d_i, d_j \in S_t$ be two messages in the data stream *S*. Let *R*(.) be a monotonically increasing function such that if d_i is deemed more important than d_j, $R(d_i) > R(d_j)$. To efficiently rank the messages, *R*(.) is applied on both these messages independently.

A tweet is considered more meaningful i) if it is retweeted more (retweet count *RT* captures the authority of the tweet [18]); and ii) if it is tweeted by a person with many followers (follower count *f* captures the authority of the user [19]). Therefore, the tweet score *R*(*d*) of a tweet *d* is computed as:

$$R(d) = \alpha RT.log f$$

Since the dynamics of an event vary at much finer time scale compared to a user's follower's count, *f* is a logarithmic factor.

The ranking function scores each arriving message efficiently, as soon as it arrives such that the more important messages are likely to be ranked higher. Please note the first criterion to choose a message is the underlying event-graph structure. The highest ranked messages associated with the *pivot nodes* in the event-graph are identified as summary (Section 5.2). Therefore a message from a user with lesser following and/or retweets would be picked in the event summary, if it is more relevant in the context of an event.

In summary, *we translate our goal to provide an event's summary into one of discovering a set of users, collectively using all the keywords in the event-topic. We show, this set of users can be* *discovered only from pivot nodes in the event-graph. We also show that pivot nodes enable us to discover informative, stable and compact summary.*

6. DISCOVERING EVENT THREADS

In this section, we present our technique to discover the contextual event summary threads capturing the evolution of live events. The evolution of an event is tracked by tracking changes in its underlying event-graph $G^t_c(V^t_c, E^t_c)$.

A possible approach for constructing the event threads is to maintain the snapshots of each event-graph in each *time window* [24], but it is not practical to construct summary threads in real time from these snapshots, as 1) instead of incrementally processing the graph, the complete event-graph needs to be processed for each event for each snapshot, thus making it impractical for processing a fast moving data stream in real time; 2) it is shown in [31], that a highly efficient mechanism is needed to keep the indexes updated for the event threads. Thus, it is not practical to create the index again for each snapshot.

Hence, we maintain contextual event threads for each event by keeping a corresponding *eventTree. EventTree* captures the evaluation of the event-graph. We assign a unique event-id (called *clusterId*) to each event-graph. When an event-graph evolves, the summary is updated and the change is recorded in its *eventTree.* With changes in event-graph, *eventTree* is maintained as follows:

Incremental changes in the event-graph: There are incremental changes in the event-graph due to addition and deletion of nodes and edges. Due to these changes, the summary may or may not change but the *clusterId* of the event-graph remains the same.

Disruptive changes in the event-graph: If the structure of an event-graph changes so much that we need to assign it a new *clusterId*, such a change is called disruptive change. For example, when two independent event-graphs with *clusterId* c_1 and c_2 merge into a single event-graph c due to emergence of new nodes/edges. The mapping $c \leftarrow c_1, c_2$ is recorded in the *eventTree.* When two event-graphs merge, their corresponding threads also merge in a single thread. Similarly, due to deletion of nodes/edges, if an event-graph c breaks into say two sub-graphs, c_1, c_2, we record $c_1 \leftarrow c$ and $c_2 \leftarrow c$ in the *eventTree.* Thus, a *eventTree* captures the evolution in the corresponding event-graph.

Whenever a disruptive change occurs, we record the parent *clusterId* (c_p)and child *clusterId* (c)relationship as an 'evolutionEdge' ($c \leftarrow c_p$ is an 'evolutionEdge'). Each 'evolutionEdge' that emerges in the current *time window w,* is processed at the end of the window, and the event summary is updated (cf. Section 6.3). We exploit the graph structure so that only the necessary 'evolutionEdges' are recorded. Note that 'evolutionEdges' track the evolution of event-graphs and they are not the edges in the graph G^t (V^t, E^t). Instead of maintaining complete snapshots, we just maintain 'evolutionEdges' to capture the differences between G^t and G^{t+1}.

We next illustrate how the event summary changes due to addition and deletion of nodes. Similar process is applicable on graph edges.

6.1 Effect of Node Addition

$G^t_c(V^t_c, E^t_c)$ is an event-graph and $G'(V', E')$ is the graph induced on G^t_c such that $E' \subset E^t_c$ be the set of *pivot edges* for graph G^t_c. When a new node (keyword) n joins the event-graph, it may induce a *new* pivot edge in graph $G^{t+1}{}_c$. A node n can join the event-graph G^t_c in two possible ways:

Case 1: Due to the addition of node n, an edge $e \in E^{t+1}{}_c - E'$ becomes a new *pivot edge.*

For example, as shown in Figure 7, when a new node n joins the cluster {A, B, C, D}, a new short cycle $n \rightarrow A \rightarrow B \rightarrow n$ is induced such that edge *AB* becomes a new *pivot edge.*

In this case, the induced graph *G'* includes an extra node (node A) and the corresponding edge(s). The existing *userCover* is extended to cover 'A'. Please note, it is possible that the event summary does not change as the existing *userCover* could be sufficient due to pivot edge *e*(B, C). This check is done in *O* (1).

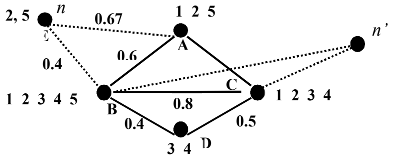


Figure 7: Change in event summary with node addition

Case 2: A node n joins the cluster such that it induces a new cycle on an existing *pivot edges* $e_p \in E'$.

In Figure 7, *n'* induces a new *short cycle* on edge BC (B→*n'*→*C*) which was already a pivot edge. Therefore, node *n'* is a peripheral node and no change is made in the event summary.

The cases underline the way event-graph is exploited to update the summary; due to the concept of pivot edges, summary does not change rapidly because of minor changes in the event-graph.

6.2 Effect of Node Deletion

A departing node breaks at least one *short cycle.* Departing node is either a pivot node or a non-pivot node.

Case 1: The departing node n is a non-pivot node (e.g., node n in Figure 7); $n \notin V'$. With the departure of node n, a *pivot edge* may no longer remain the *pivot edge.* Since n is not a pivot node it can impact only one pivot edge as it induces only one cycle. However, the *userCover* continues to remain a valid user cover as the edge $e_{(n', u)}$, erstwhile *pivot edge*, remains part of the event-graph $G^{t+1}{}_c$.

Case 2: The departing node n is a *pivot node* for *edge e* (*n, v*). For example, consider node B in Figure 7. Departure of a *pivot node* has a significant impact on the event-graph G^t_c and the graph may break into multiple sub-graphs or it may get dissolved if it no longer possesses the *short-cycle* property (in that case, the event ceases to exist as a live event). Each surviving sub-graph must possess SCP. Each of the surviving sub-graphs is assigned a new *clusterId.* The *updateEvolutionEdge* () records the relationship between the old and each new *clusterId.* For each 'evolutionEdge' $c_i \leftarrow c$, we extract the event-graph $G^{t+1}{}_{ci}$, update its summary and its event thread.

```
Algorithm: UpdatePECover (node n)
$\forall n' \in N(n)$ /*N(n) returns neighbors of n in $G^t_c$*/
  If $n' \in V'$ /* V' is a set of pivot nodes in induced graph G'*/
    $\forall u \in N(n')$
      if $e_{(n', u)} \in E'$
        if $e_p$ is no longer a pivot edge
          $E' \leftarrow E' - e_{(n', u)}$;
      else /*edge is a pivot edge*/
        childId ← getNewClusterId();
        updateEvolutionEdge (childId, clusterId);
```

The merging and splitting of event-graphs, results in an *eventTree* which is a Directed Acyclic Graph (DAG), representing the contextual event summary thread similar to shown in Figure 1.

6.3 Temporal Evolution of Event Thread

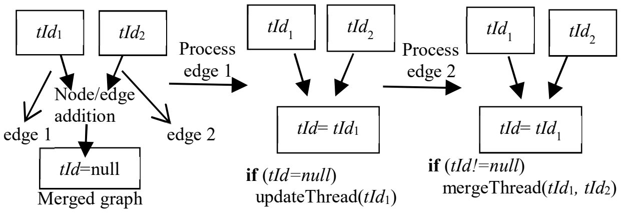


Figure 8: Processing 'evolutionEdges' to update event threads

For all the 'evolutionEdges' (*childId←parentId*), the corresponding event threads are updated. Figure 8 depicts how the event threads evolve with time. Each *eventTree* is identified by a unique id, called *tId*. For each *childId←parentId* 'edge', if the parent cluster's *tId* is null, it is a new event. We initialize its *eventTree* and identify event summary (using approxUserCover). Otherwise, we merge the *eventTree* of the parent cluster with the *tId* of the child cluster and update the event summary. The evolving event summary is recorded in the event thread. Similarly, event threads are updated in case of event-graph split. For two *clusterIds* c_1 and c_2 in an *eventTree,* if c_1 is ancestor of c_2, c_1 has occurred before c_2 in real world time. Each path in an event thread, from each of its roots to each of its leaves, exposes a different facet of the event. The event thread in Figure 1 has 6 facets.

7. PERFORMANCE EVALUATION

The goals of our experiments are to study our system's ability a) to construct informative, complete, meaningful, stable and compact event summaries in real time (Section 7.2); b) to discover the contextual event summary threads in real time (similar to Figure 1) and to study the impact of the changes in the granularity of the event-graph on the event summary and event threads (Section 7.3); c) to discover event threads efficiently and in a scalable manner (Section 7.4). The experiments use the prototype built by us [31] and run on a quad-core 2.61 GHz, 4GB RAM machine running Windows 8 and Java as programming language.

In Table 2, we describe the Twitter traces used in experiments. We use two types of traces: a) general timeline based (ALL) which contains all the tweets generated within US geography within a time window, provided by Twitter API and b) event specific (ES) traces. The event specific traces were created as follows: For each event, we specify a set of rules. Each rule contains one or more relevant keywords and/or hashtags associated with the event. Any tweet, containing the keywords from a rule is included in the event trace. We have carefully selected the traces to cover the entire spectrum of event change density -- from very low (22) to very high (775). **Events change density** specifies total number of times *all* the underlying event-graphs in a trace evolve every 100k tweets. Traces are read in their chronological order to mimic the real-time arrival of the tweets.

Table 2: Details of Datasets

Event	*#of Tweets*	*Events change density*
Big Data	200k	775 changes/100k tweets
Nairobi–complete	720k	274 changes/100k tweets
Syria	1.7million	182 changes/100k tweets
Twitter Time Line (ALL)	3.2million	22 changes/100k tweets

7.1 Discovering Base Events

To the best of our knowledge, no previous system exists that summarizes a complete live data stream in the absence of any user query. Therefore, we construct the ground truth as follows: Live events unraveling in the data stream are the base events and form the ground truth for our system. The objective of our experiments is to study the performance of our summarization technique and its ability to construct meaningful *contextual summary threads* for the events given to it as ground truth. Any algorithm discovering dense graphs as event-topics in a highly dynamic graph can be used to provide the base events. For our experiments, we use the algorithm in [15] – it efficiently discovers the events in a live data stream with high precision and recall. It is shown in [15] that the events discovered by this system correlates highly with real world events reported in Google news headlines. Additionally, it discovers many other real world events that do not occur in Google news headlines.

The number of event-topics as well as the number of keywords in an event-topic in a data stream depends on the dynamic event-graph $G^t (V^t, E^t)$ at time *t*. The set of nodes V^t in the graph G^t depends on burstiness threshold γ and the set of edges E^t depends on the edge similarity threshold λ. The default values are: γ=5 and λ=0.2, unless specified otherwise. The message block size *m* is set to 1000 and *w* (cf. Table 1) is set to be 75 for all the experiments.

7.2 Quality of Event Summarization

Informative-ness: We identify the *userCover* only for the pivot nodes in an event-graph (Section 5). The premise is that the important keywords in the event-graph are likely to be pivot nodes. The peripheral nodes in the event-graph may or may not be covered in the event summary. We quantify the informative-ness of the summary as follows: If there is a keyword in an event-topic that is a proper noun and is not present in the event summary, we count it towards loss of precision. Precision is computed as the fraction of proper noun keywords present in the event summary among all the proper noun keywords in the event-topic. Let there be N events in a given trace. Let R_i be the set of proper noun words in the summary of the i^{th} event discovered by our algorithm; $1 \leq i \leq$ N. Let B_i be the set of all the proper noun keywords present in the event-topic of event *i*. The precision of informative-ness, P_I, is defined as:

$$P_I = \frac{\sum | R_i \ |}{\sum | B_i \ |} ;1 \leq i \leq N$$

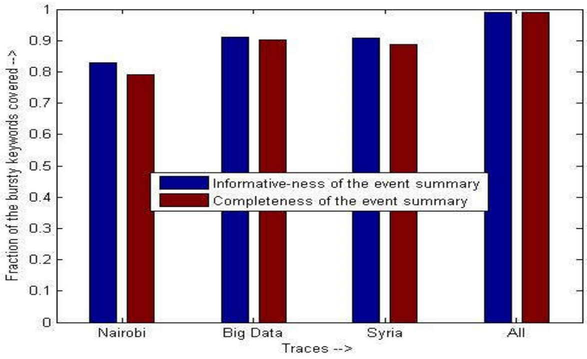


Figure 9: Summary Informative-ness and Completeness

Complete-ness: We compute the fraction of *total* keywords in the event-topic covered in the event summary to compute the completeness score P_C. As shown in Figure 9, P_C for various traces varies from 79% to 99%. P_C is marginally lower than P_I, as typically noun keywords are more central in the event-graphs. For the ALL trace, since almost all the keywords are covered in the event summary, there is no difference in the two scores. We see, *that our system achieves very high informative-ness and complete-ness score.*

Stability: We study the stability of the event summaries by our algorithm for live events. The results are shown in Table 3. Since different traces have different sizes, we report the results in terms of event change density/100k tweets.

Table 3: Rate of changes in event-graphs for different traces

Total event-graph changes/ 100k tweets	**Changes (addition in event-graph)**	**Changes (event-graph break)**	**No change in summary (additions)**	**No change in summary (deletions)**
Big data-775	114	33	411	217
Nairobi-274	43	5	173	53
Syria-182	30	3	114	35
ALL- 22	1.5	0	13.5	7

We see that across the traces, more than 80% of the changes in the event-graphs do not result in any change in the summary. For example, for Big-Data trace, for every 775 changes in the event-graphs; event summary remains the same for 411 changes when nodes/edges or messages get added to the event-graph and for 217 changes when a node/edge gets deleted from the event-graph, i.e., no summary change for 81% of event-graph changes. Thus, *the event summary remains stable for a large fraction of changes.*

Summary-size: Next, we compare the message pool size of an event with the number of messages in its summary (summary-size). The average message pool size varies from 74 tweets (for ALL trace) to 1394 tweets (for Syria trace) as shown in Figure 11, denoting that for an emerging event in the Twitter data stream, a large number of related messages are posted. The number of messages is per event-graph and not per event thread (an event thread captures the evolution of associated event-graph(s)). We divide the events based on the number of keywords in the event-topic, as shown in Figure 10 and Figure 11. For ALL trace, no event-topic contained more than 16 keywords.

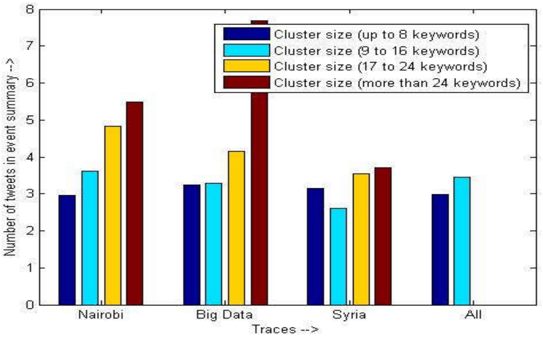


Figure 10: Average number of tweets in 'event summary' for different event-topics sizes

As expected, summary-size increases with increasing event-topic size (Figure 10). Similarly, number of messages pertaining to an event increases with event cluster size (Figure 11). The key insights are:

1) Summary-size is independent of the message pool size (i.e., tweets associated with an event-graph). It depends only on the underlying information. For example, for Syria trace, average message pool size increase from 128 to 1394 for different size event-topics but the summary-size remains almost stable. The event summaries were highly meaningful.

2) Average summary-size varies from 2.61 to 7.71 tweets for different traces for event-topics comprising up to a few hundred tweets. Thus, our system discovers highly compact summaries.

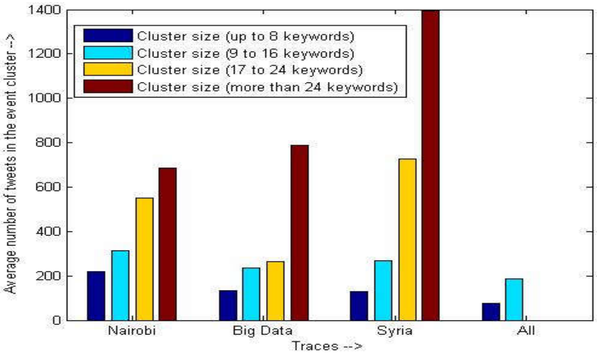


Figure 11: Average number of tweets in the event 'message pool' for different sizes of event-topics

We construct alternative summary for a discovered as follows: For an event-graph, we randomly select a message from the message pool covering an event keyword. We keep on selecting messages till each keyword in the event graph is covered by at least one message. This 'naïve method' serves as a baseline to compare the performance of our system.

Since we select the messages randomly in 'naïve method', qualitatively, the summary by naïve method was markedly inferior for almost all the events. The other issues with this naïve method were; 1) A significant number of redundant messages occur in the event summary. For many events, the summary size was more than twice the summary discovered by our system.; 2) the naïve method will not discover the event threads, exposing different event facets.

To further compare the quality of event summary we compared the summary discovered based on our approach with Google News headlines (for the events which also appear in Google headlines). In Table 4, we present the comparisons between a few of the Google headlines with our summary tweets.

Table 4: Google News Headlines Vs. Event Summary discovered by our approach

Google Headline	**Summary based on our approach**
India vs New Zealand, 3rd Test: Ashwin 6/81 hands India huge first innings lead	India vs New Zealand, 3rd Test: Ashwin 6/81 hands India huge first innings lead https://t.co/ih5oTkOlIY
NASA Mission Tests Thrusters On Journey To Asteroid	NASA probe tests thrusters on journey to asteroid Bennu - Zee News: NASA probe tests thrusters on jou... https://t.co/2Tv0XC71cJ
Pampore attack: Militants holed up inside govt building; combing operations intensify	RT @kashmirglobal: Smoke and dust engulf a government building where suspected militants have fighting with Indian forced in Pampore…
NASA resupply mission to space station postponed	Atlantic Storm System Delays NASA Resupply Launch to Space Station via NASA https://t.co/wdWmqwhz7k
Obama pushes NASA to send humans to Mars by 2030s	Can the U.S. Really Get Astronauts to Mars by 2030?: President Obama renewed his call to send Americans to th... https://t.co/ZeSADfvDNZ
1000 asteroids heading towards Earth; conspiracy theorists claim end of the world is near!	RT @Ufo_area: Asteroid mission: 1000 space rocks heading towards Earth – Daily Star https://t.co/ExDVSJOzly #Asteroid

Microsoft Office for Android will be supported on Chrome OS	Microsoft Office for Android will be supported on Chrome OS +AC0- The Indian Express https://t.co/7xLBJJpSKW

We see that the summary tweets represent the Google headlines very accurately. For a few headlines, Google headline matches completely with the tweet selected by our approach. However, the timestamp of tweets in our summary is ahead by few minutes to few hours, for different event, compared to their corresponding appearance in Google headlines. Details are omitted due to lack of space.

In our next experiment, we study the performance of our system to expose context event summary threads.

7.3 Contextual Event Summary Threads

How do Real-time Events Evolve with Time? – Next, we study what fraction of the real-time events evolve into event threads. We divide the events into; 1) standalone events, i.e., events which do not result in threads and 2) event threads. We plot the density of events per 100k tweets for each trace. The results are shown in Table 5. The event density is highest for Big-data trace. We see; 1) a significant number of events result in event threads; and 2) the density of events is much higher for event specific traces as opposed to ALL trace. ALL trace has a density of 1.1 standalone events and 1 event thread/100k tweets respectively. Since the average tweets/event is highest for Syria trace (Figure 11), the density of events for it is relatively smaller.

We show the average number of times an event-graph changes during its life cycle in Table 5. An interesting insight is that even though the density of events is lowest for ALL trace, the average number of times its event-graphs changes during their life span is significantly higher compared to ES traces. The reason is, density of tweets related to a single event in ALL trace is very low. Therefore, the changes in the existing event-graphs are only marginal and the event-graphs absorb such changes, resulting in a longer life span. In summary, *a large fraction of events result in threads. Further event threads evolve only when there are significant changes in the event.*

Table 5: Event density and Average number of times an event-topic changes during its life span

Density/100K Tweets	Nairobi	Big Data	Syria	All
Density of Standalone Events	19.44	94.38	16.25	1.1
Density of Event Threads	39.9	59.9	15.6	1
# of Changes in an event-topic during its life span	6.22	5.88	6.46	10.88

How Complete are the Event Threads? - In this experiment, we study how the real-time events evolve. An event thread is a DAG. The depth of a DAG is the length of the path from its root to its deepest leaf (depth of DAG in Figure 1 is 9). To count the number of facets in the DAG, we sum the total number of unique paths from the root(s) to each of the leaf nodes of an event thread. We study the temporal evolution of the events by computing the average depth and average number of facets of the event threads. The facets are counted as: $facets = \sum_{l \in L} \sum_{r \in R} p_l^r$ where p_l^r is total number of paths to reach from root node r to leaf node l and L (R) is the set of leaf (root) nodes in the event-graph. Depth of an event thread and its facets capture the complexity of the events. The results for different datasets are shown in Table 6. Only the event threads, not the stand-alone events, are considered for this experiment.

Table 6: Average depth and average facets for an event

Event Thread Complexity	Nairobi	Big Data	Syria	All
Average event depth	5.98	4.97	5.42	2.77
Average event facets	2.63	2.25	2.27	1.21

Average depth/facets of the event threads are highest for Nairobi trace at 5.98/2.63 and lowest for ALL trace at 2.77/1.21. Hence, event complexity is highest for Nairobi trace. The result signifies that our algorithm can handle changes in a fast- moving data stream gracefully. For the default values of λ and γ, the maximum depth of an event was 30 (for Nairobi trace) with 9 facets, exposing the complex way the live events evolve.

How does Granularity of Event-Graph impact the summary? To vary the granularity of the dynamic graph G^t *(V^t, E^t),* we vary the burstiness threshold (Bt) γ and edge correlation threshold (Ec) λ. If γ and/or λ are reduced, there will be more nodes and/or edges in the graph G^t *(V^t, E^t),* leading to more events being discovered and vice versa. In Figure 12, we show how the number of event threads (every 100k tweets) varies -- by varying γ and λ for different traces.

We find two distinct trends: 1) for Big-Data and Syria traces, with more nodes/edges in the dynamic graph G^t (lower γ and/or λ), the number of event threads increase but the depth and the facets are not significantly impacted; 2) for Nairobi and ALL traces, with more nodes/edges in G^t, the number of events is not impacted much but the event depth/facets increase, exposing the same events at finer granularity. In summary, *when the messages related to an event tend to come in bursts, we observe trend* (1). *If the messages related to an event are distributed more evenly in the data stream, we observe trend* (2).

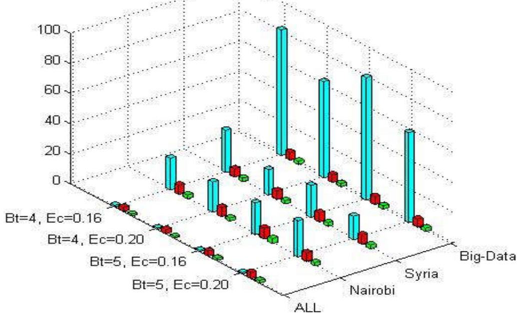


Figure 12. Event Density (blue), event thread depth (red) and event thread facets (green) with varying γ (Bt), λ (Ec)

7.4 Efficiency and Scalability

In this experiment, we analyze the overhead of our method to summarize the events and arrange them into contextual summary threads. We also analyze the scalability of our approach, i.e., how many tweets are processed/second (TPS). We discover the event-topics in a twitter trace without contextual summarization system and with it. In Table 7, we show the overhead of our system. We see that the summarization algorithm imposes only a marginal overhead over base event discovery system in [15]. Overhead of our method is highest for Nairobi trace at 12.36% and smallest for ALL trace at 3.17%. The tweet processing rate is highest for ALL trace at 6631 TPS and lowest for Big-Data trace at 1044 TPS, with summarization system on. For Nairobi trace, the rate is 4473 TPS with summarization and 5026 without summarization.

Table 7: Tweet processing rate per second (TPS)

TPS	Nairobi	Big Data	Syria	All
With Summarization	4473	1044	2505	6631
Without Summarization	5026	1129	2614	6841

Without the summarization, the TPS for ALL trace is 6841 for the algorithm presented in [15]. ALL trace is closest to the general Twitter data stream. The average rate for global Twitter data stream is reported to be 5700 TPS in August 2013 (peak rate is ~144k TPS) [26]. Therefore, *our technique is highly scalable for real time processing and does not impose a big overhead to identify event summary and event threads* over event discovery algorithm.

In summary, we present a novel system that constructs meaningful, stable, and compact event summaries for the events present in an unfiltered data stream. We also discover contextual event threads in real time over live data streams efficiently.

8. CONCLUSION

In this paper, we present a novel unsupervised technique that builds the summaries for emerging events in real time in a complete fast moving data streams in absence of any user query. The summaries are complete and meaningful and contain the informative messages for the underlying events. Our technique also discovers the contextual event summary threads in a scalable manner. It is not necessary that the most recent messages are also the most informative for a live event. However, the most informative messages about the event are present in its summaries. We plan to extend our technique to enable improved real time search over data streams.

9. ACKNOWLEDGMENTS

Pranjal Khare (student, Department of CSE, IIT-Bombay) used our system to produce results presented in Table 4.

10. REFERENCES

[1] Ramesh Nallapati, Ao Feng, Fuchun Peng, and James Allan, “Event threading within news topics”, in CIKM, 2004.

[2] Ao Feng, and James Allan, “*Finding and linking incidents in news”*, CIKM 2007, page 821-830.

[3] W. Gao, P. Li, K. Darwish, “Joint Topic Modeling for Event Summarization across News and Social Media Streams”, in CIKM 2013.

[4] Chen Lin et al., “Generating Event Stroylines from Microblogs”, in CIKM 2012, page 175-184.

[5] L. Shou, Z. Wang, K. Chen, G. Chen, “Sumblr: Continuous Summarization of Evolving Tweet Streams”, in SIGIR 2013.

[6] Yajuan Duan et al., "An Empirical Study on Learning to Rank of Tweets", in COLING 2010.

[7] J. Vosecky, D. Jiang, K. W. Leung, W. Ng, “Dynamic Multi-Faceted Topic Discovery in Twitter”, in CIKM 2013.

[8] X. Yang, A. Ghoting, Y. Ruan, S. Parthsarthy, “A Framework for Summarizing and Analyzing Twitter Feed”, in SIGKDD 2012.

[9] K. Kutzkov, R. Pagh, "Triangle counting in dynamic graph streams", in SWAT 2014.

[10] B. Sharifi, M. Hutton, J. Kalita,. “Summarizing Microblogs Automatically”, in NAACL-HLT, 2010.

[11] D. Chakrabarti, K. Punera, “Event Summarization using Tweets”, in ICSWM 2011.

[12] Z. Yang, K. Cai, J. Tang, L. Zhang, Z. Su, J. Li, “Social Context Summarization”, in SIGIR 2011, pages 255-264.

[13] F. T. Chua, S. Asur, “Automatic Summarization of Events from Social Media”, ICWSM 2013.

[14] M. Mathioudakis, N. Koudas, “TwitterMonitor: Trend Detection over the Twitter Stream”, SIGMOD 2010.

[15] M. K Agarwal, K. Ramamritham, M. Bhide "Real Time Discovery of Dense Clusters in Highly Dynamic Graphs: Identifying Real World Events in Highly Dynamic Environments", in VLDB 2012.

[16] Bansal N., Chiang F., Koudas N., Tompa F. “Seeking Stable Clusters in the Blogosphere”, in VLDB 2007.

[17] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment", in Journal of the ACM (JACM), Vol. 46, 1999.

[18] H. Kwak, C. Lee, H. Park, S. Moon, “What is Twitter, a Social Network or a News Media?”, in WWW 2010.

[19] C. Chen, F. Li, B. C. Ooi, S. Wu, “T1: An Efficient Indexing Mechanism for Real-Time Search on Tweets”, in SIGMOD 2011.

[20] X. Hu, Y. Tao, C.-W. Chung, “Massive Graph Triangulation”, in SIGMOD 2013.

[21] S. Brin, L. Page, "The Anatomy of a Large-Scale Hyper textual Web Search Engine", J. of Computer Networks, Vol. 30, 1998.

[22] U. Brandes, "A Faster Algorithm for Betweenness Centrality", J. of Mathematical Sociology, 25(2), 163-177, 2001.

[23] J. Xu, D. Kalashnikov, S. Mehrotra, “Efficient Summarization Framework for Multi-Attribute Uncertain Data”, in SIGMOD 2014.

[24] P. Lee, L. Lakshmanan, E. Milios, “Incremental Cluster Evaluation Tracking from Highly Dynamic Network Data”, in ICDE 2014.

[25] http://www.nada.kth.se/~viggo/wwwcompendium/

[26] https://blog.twitter.com/2013/new-tweets-per-second-record-and-how

[27] A. Pavan, et al., “Counting and Sampling Triangles from a Graph Stream”, in VLDB 2014.

[28] N. Wang et al., "On Triangulation-based Dense Neighborhood Graphs Discovery", in PVLDB, 4(2):58–68, 2010.

[29] http://en.wikipedia.org/wiki/Chordal_graph

[30] https://en.wikipedia.org/wiki/Westgate_shopping_mall_attack

[31] M. K. Agarwal, D. Bansal, M. Garg, K. Ramamritham, “Keyword Search on Microblog Data Streams: Finding contextual Messages in Real Time”, in EDBT 2016.

An Effective and Efficient Truth Discovery Framework over Data Streams

Tianyi Li
School of Computer Science
and Engineering
Northeastern University
Shenyang, China
litianyi_neu@163.com

Yu Gu
School of Computer Science
and Engineering
Northeastern University
Shenyang, China
guyu@mail.neu.edu.com

Xiangmin Zhou
School of Computer Science
and Information Technology
RMIT University
Melbourne, Australia
xiangmin.zhou@rmit.edu.au

Qian Ma
School of Computer Science
and Engineering
Northeastern University
Shenyang, China
maqian_neu@163.com

Ge Yu
School of Computer Science
and Engineering
Northeastern University
Shenyang, China
yuge@mail.neu.edu.com

ABSTRACT

Truth discovery, a validity assessment method for conflicting data from various sources, has been widely studied in the conventional database community. However, while existing methods for static scenario involve time-consuming iterative processes, those for streams suffer from much sacrifice on accuracy due to the incremental source weight learning. In this paper, we propose a novel framework to conduct truth discovery over streams, which incorporates various iterative methods to effectively estimate the source weights, and decides the frequency of source weight computation adaptively. Specifically, we first capture the characteristics of source weight evolution, based on which a framework is modeled. Then, we define the conditions of source weight evolution for the situations with relatively small unit and cumulative errors, and construct a probabilistic model that estimates the probability of meeting these conditions. Finally, we propose a novel scheme called adaptive source reliability assessment (ASRA), which converts an estimation problem into an optimization problem. We have conducted extensive experiments over real datasets to prove the high effectiveness and efficiency of our framework.

CCS Concepts

•**Database Management** → **Database Applications;**

Keywords

truth discovery; data streams; source reliability; data quality

1. INTRODUCTION

The current big data era has witnessed various sources providing information on the same set of objects or events [18]. The data inconsistency across multiple sources is an important research issue in many applications. The real world applications like weather situation analysis and health-care require techniques to identify which data sources are more reliable or what information is accurate. For example, when we identify the weather condition of a city, the inconsistent information may be obtained from multiple websites. As another example, different medical records on a patient may be found from different hospitals. Thus, it is highly demanded to automatically identify trustworthy information from conflicting data. For this task, truth discovery has been proposed to model the source quality and derive the truth based on a principle: the information from a reliable source is trustworthy and the source providing trustworthy information is reliable. By leveraging this principle, several mechanisms have been proposed in previous works for both static and dynamic data.

Consider a set of conflicting stock information for Apple Inc. at certain time as shown in Figure 1. As the information on the open price is arriving continuously, the truth on it evolves over time. In addition, the value from Insidestocks is closer to the truth at t_{i-1}, while that from Stocksmart is closer to the truth at t_i. This implies the reliability degrees of these three sources change over time as well. Thus, it is vital to identify the reliability of sources and the truths over continuous data streams, and develop advanced techniques for the truth discovery under dynamic scenario. Existing approaches for truth discovery mainly focus on static data [6, 7, 8, 19, 2, 22, 3, 1, 15, 5, 9, 12, 4, 14, 24], where an iterative process is exploited. The truth discovery process constantly iterates until the source weight converges to an optimal value. Applying the iterative process to the truth discovery at each timestamp over streams, the high accuracy performance can be achieved. However, these approaches suffer from expensive time costs, which is not applicable to high-speed data streams. Recently, some approaches have been proposed to improve the truth discovery efficiency by learning source weights and deriving truths incrementally [11, 23]. However, these methods sacrifice much accuracy, because they model each source weight as a constant. The reliability of each source estimated by them is converged to a value, while the true source weights in real applications are constantly changing over time [16].

To effectively and efficiently discover truths over streams, we need to well address three issues. First, various iterative methods should be incorporated in a nice way to find the truths and the reliability of sources. This is important, as the optimal truths and source weights at each timestamp can only be derived by iteration strategy. As a result, the accuracy of truth discovery over data streams can be improved. Second, we need to design a set of advanced techniques which adaptively decide the frequency of source weight assessment to minimize the number of iterative operations. As data streams flow in large volume at high speed, it is clearly unacceptable to perform iterations at each timestamp. Finally, we should study the errors caused by not accessing the source weights continually over streams, and control these errors in a certain range.

In this paper, we propose a novel framework for effective and efficient truth discovery over streams. The idea behind it is to incorporate the iterative process in truth discovery for high accuracy and adaptively reduce the frequency of source weight assessment for high efficiency. Specifically, we first define two concepts, *Unit error* and *Cumulative error*, to describe the error caused by not changing the source reliability over data streams. Then, we present the relationship between each of these two concepts and the source reliability change based on theoretical analysis, which guarantees the accuracy of our truth discovery framework. For minimizing the source weight assessment frequency, we turn the problem of source weight assessment into an optimization problem and propose a scheme called ASRA to determine this frequency adaptively over data streams. In summary, we make the following contributions:

- We speculate the condition of the source reliability evolution under the constraints of small errors based on theoretical analysis, which guarantees the accuracy of our method. A probabilistic model is constructed to estimate the probability of meeting these conditions.
- We propose an optimization-based scheme ASRA, that minimizes the source reliability assessment frequency by estimating the maximum value of cumulative error smaller than a given threshold in a certain confidence level of probabilities.
- We propose a framework, which adaptively determines the time of source reliability assessment by combining the incoming data. Our framework incorporates various iterative approaches to estimate the reliability of sources, and balances the efficiency and accuracy by tuning the parameters.
- We validate the proposed framework on real datasets, and the results demonstrate the high performance of our proposed framework in term of effectiveness and efficiency.

The rest of paper is organized as follows. We survey the related work in Section 2, and formulate the research problem in Section 3. Section 4 proves some conclusions of truth discovery over data streams. Section 5 introduces the probability model and proposes our method. Section 6 conducts experiments and analyzes experimental results. Section 7 concludes our paper.

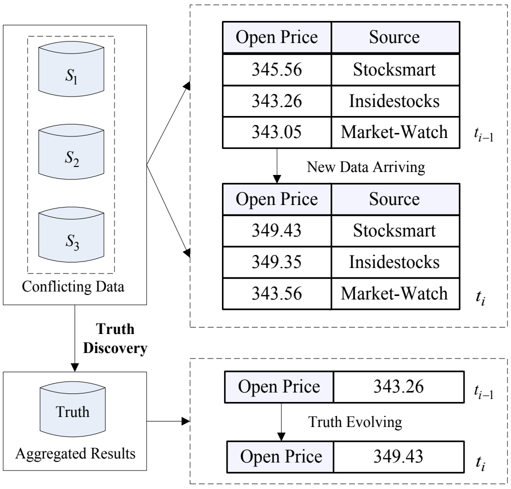


Figure 1: An Example of Truth Discovery over Data Streams

2. RELATED WORK

Truth discovery has been widely recognized in research community, and applied in several domains such as social sensing [17], health communities [13] and wireless sensor networks [20]. Previous works on truth discovery mainly focus on static databases [8, 7, 19, 6, 2, 22, 3, 1, 15, 5, 9, 12, 4, 21, 24]. In [19], Yin et al. propose an algorithm called TruthFinder that identifies truths using an iterative process. In [6], Galland et al. propose three alternative fixpoint algorithms, Consine, 2-Estimates and 3-Estimates, to estimate the truths and the reliability of sources. In [22], Zhao et al. study the truth discovery problem by modeling the two-sided source quality and leveraging Gibbs sampling. In [21], a probabilistic model is designed for the truth discovery over numerical data. In [8], an optimization-based framework is proposed to resolve the conflicts among multiple sources of heterogeneous data types. A confidence aware truth discovery method is proposed to find truths from the conflicting information with long-tails phenomenon [7]. However, none of these approaches can be directly applicable to data streams due to the costly iterative process.

Source correlation analysis has been studied as another topic of truth discovery [2, 3, 1, 15, 5, 9]. In [2], the ACCU model is proposed, which applies Bayesian analysis to decide the dependence between sources. In [3], Dong et al. propose a probabilistic-based approach to decide the copying relationship in a dynamic world. A Hidden Markov Model (HMM) is utilized to decide whether a source is a copier of another source and identify the specific moments at which it copies. In [1], a global model is proposed to identify the co-copying and transitive copying relationships. In [15], Pochampally et al. explore the correlation beyond copying, and propose a Bayesian-based model for addressing the positive and negative relationships in sources. A multilayer probabilistic model is proposed to compute the trustworthiness levels of sources [5]. A set of experiments is conducted to analyze the advantages and limitations of several truth discovery methods [9].

Recently, some attempts have been conducted to solve the truth discovery problem over data streams. In [23], Zhao et al. propose a probabilistic model that handles conflicting values over data streams. However, their method

Table 1: Notations

Notation	Definition	Defined in (Section)
$v_i^{(k,e,m)}$	the observation of the m^{th} property for the e^{th} object by the k^{th} source at t_i	3
V_i	the observations of all the objects on all the properties from all the sources at t_i	3
w_i^k	the weight of the k^{th} source at t_i	3
W_i	the source weight collection at t_i	3
$v_i^{(*,e,m)}$	the truth of the m^{th} property for the e^{th} object at t_i	3
V_i^*	the truths of all the objects on all the properties at t_i	3
λ	the smoothing factor	3.1
Δw_i^k	the source weight evolution on k^{th} source at t_i	3.2
ε	the unit error threshold	4
α	the probability threshold	5.2
E	the cumulative error threshold	5.2

can only work over categorical data. In [11], Li et al. proposed an incremental truth discovery method by transforming their optimization-based solution into a probabilistic model. However, the previous truth discovery work has shown that true source weights change over time [16], and this key point has not been considered in the models proposed in [23] and [11]. The source weight learned by these incremental methods converges to a certain value, which is considered as the corresponding true source weight. Although a smoothing factor has been introduced to capture the source's reliability changes [11], the source weight computed by it also finally converges to a certain value. Thus, these incremental methods suffer from low accuracy compared with optimization-based solutions. To the best of our knowledge, our work is the first attempt ever made to trade off the accuracy and efficiency of truth discovery over streams flexibly by tuning the parameters [10]. Moreover, with our proposed framework, various iterative truth discovery algorithms can be utilized to improve accuracy with neglectable efficiency losses. The notation used in this paper is listed in Table 1 for easy reference.

3. PROBLEM FORMULATION

In this section, we illustrate our proposed framework for truth discovery over data streams. Before proceeding to the problem formalization, we will introduce several important concepts first, *Observation*, *Source Weight*, and *Truth*.

Definition 1. An *observation* is the data that describes an object property of a source at a timestamp. We denote the observation of the m^{th} property on the e^{th} object from the k^{th} source at t_i as $v_i^{(k,e,m)}$, and all observations at t_i as V_i.

Definition 2. A *source weight* is the reliability degree of a source at a timestamp. The source weights at t_i are denoted as $W_i = \{w_i^1,\ w_i^2,\ \ldots,\ w_i^K\}$, where w_i^k is the reliability degree of the k^{th} source at t_i.

Definition 3. A *truth* is an aggregated result derived from truth discovery. We denote the truth of the m^{th} property for the e^{th} object at t_i as $v_i^{(*,e,m)}$. Let $v_{o,i}^{(*,e,m)}$ be the *optimal truth* satisfying the convergence criterion of a given iterative method at t_i, and $Dist$ be a distance function. Given a timestamp t_k for source weight assessment, the truth $v_k^{(*,e,m)}$ is a value that holds the condition: $Dist(v_{o,k}^{(*,e,m)}, v_k^{(*,e,m)}) = 0$. Given a timestamp t_j without source weight assessment, and two thresholds, ε, α, the truth $v_j^{(*,e,m)}$ is a value that is derived by previous source weights W_i $(i < j)$ and holds the condition: the probability of $Dist(v_{o,j}^{(*,e,m)}, v_j^{(*,e,m)}) \le \varepsilon(j-i)^2$ is no less than α. The truths of all the objects on all the properties at t_i are denoted as V_i^*.

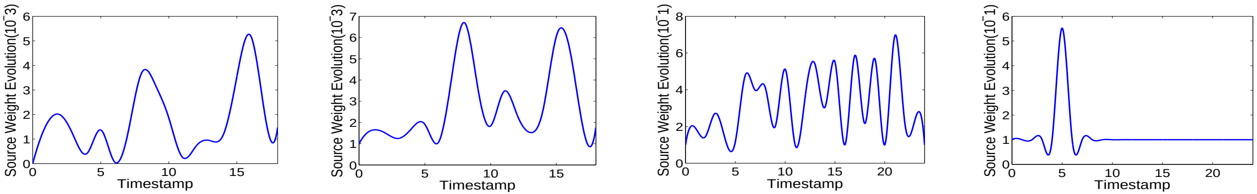
Given a set of observations V_i, truth discovery over data streams is to automatically infer the truths V_i^* and the source weights W_i at each timestamp t_i. In this paper, we propose a novel framework that balances the effectiveness and efficiency of truth discovery over data streams. The idea behind it is to incorporate iterative process in truth discovery for high accuracy and adaptively determine the frequency of source weight assessment for high efficiency. For this task, we first formalize the truth computation and the source weight evolution to analyze the error caused by not assessing source weights continually over data streams. Then, we define two concepts, unit error and cumulative error, and speculate the relationship between the source weight evolution and the two errors based on theoretical analysis, which guarantees the accuracy of our framework. Finally, we propose an optimization-based scheme which minimizes the iterative operations, and then propose our method which adaptively decides the source weight assessment frequency by combining the incoming data. We denote the timestamp that our method updates the source weights as *update point*. Next, we will introduce our basic ideas on truth computation and source weight evolution.

3.1 Truth Computation

Truth computation is to keep the truths close to the claims from reliable sources. Traditional voting or averaging schema assumes all sources are equally reliable, which is generally unreasonable in real applications. To overcome this problem, many truth discovery methods use weighted voting or averaging to obtain the truths [8, 7, 11, 19, 6, 2], which makes the observations from high quality sources more important. In this paper, we infer the truth by exploiting the same weighted averaging strategy considering its advantages:

$$v_i^{(*,e,m)} = \frac{\sum_{k=1}^{K} w_i^k \cdot v_i^{(k,e,m)}}{\sum_{k=1}^{K} w_i^k} \tag{1}$$

According to this weighted combinations, the information from the higher quality sources is more trustworthy, which is consistent with the principle of truth discovery. However, for truth discovery over data streams, the information



(a) Stock Dataset (S_1) (b) Stock Dataset (S_2) (c) Weather Dataset (S_1) (d) Weather Dataset (S_2)

Figure 2: Source Weight Evolution in Real-World Applications

usually evolves smoothly. To capture this characteristic, we add one smooth constraint on the aggregated results. As such, the truth $v_i^{(*,e,m)}$ is computed by:

$$v_i^{(*,e,m)} = \frac{\sum_{k=1}^{K} w_i^k \cdot v_i^{(k,e,m)} + \lambda \cdot v_{i-1}^{(*,e,m)}}{\sum_{k=1}^{K} w_i^k + \lambda} \tag{2}$$

where λ is the smoothing factor [11]. This equation treats the truth $v_{i-1}^{(*,e,m)}$ as the information from a pseudo source and λ as the weight of this source.

Existing iterative truth discovery methods usually assess the truths and source weights by conducting an alternating iterative process [8, 7, 11, 19, 6, 2]. In other words, such methods update truths while fixing source weights and then update source weights while fixing truths until convergence. We aim to design a framework which can embed various iterative truth discovery approaches for the accuracy improvement, and infer the truth by exploiting the weighted combinations strategy (i.e., Formula (1) or (2)). Thus, an iterative truth discovery method can be plugged into our framework only in the case that its truth computation is in the form of weighted combinations.

3.2 Source Weight Evolution

Based on the principle of truth discovery, the source weight reflects the contribution of a source to the results of weighted combinations. Therefore, a relatively smooth evolution of a source weight implies a small variation on the contribution of this source. Under this situation, neglecting the updating of source weights will cause small errors, while decrease the iterative process. Thus the iterative methods can be applied to dynamic scenarios. The *Source Weight Evolution* Δw_i^k on k^{th} source at time t_i is computed by:

$$\Delta w_i^k = \left| w_i^k / \sum_{k=1}^{K} w_i^k - w_{i-1}^k / \sum_{k=1}^{K} w_{i-1}^k \right| \tag{3}$$

To observe the evolution of source weights, we conduct a set of experiments on two real-world datasets: Stock Dataset and Weather Dataset. These datasets have been used in the evaluation of truth discovery solutions [9, 3], and their ground truths are available. For each dataset, we randomly select two sources, S_1 and S_2, for tests. Each source weight is quantified by comparing its observations with the ground truths and measuring the closeness between them. Since data usually contain multiple attributes in real applications, we normalize the deviation from various attribute values. Figure 2 shows the experimental results on source weight evolution over two different real-world datasets. Clearly, the evolution of source weights is quite minor at some moments. Under this scenario, it is natural to utilize previous source weights instead of current ones to obtain truths. For one thing, since the source weight computation is neglected, the iterative process is decreased under dynamic scenario. Thus, the iterative methods are applicable to data streams to improve the accuracy of truth discovery. For another, the deviation between the optimal truth and the approximate one will be small as well. Next, we will analyze this deviation caused by un-assessing source weights.

4. THEORETICAL ANALYSIS

In this section, we prove the condition of the source weight evolution under the constrains of small errors caused by un-assessing source weights. We first define the error in the form of mathematical formula. The unit error Φ_j^i $(i < j)$ is given by:

$$\Phi_j^i = \left(\frac{v_{o,j}^{(*,e,m)} - v_{i/j}^{(*,e,m)}}{v_j^{(\max,e,m)}}\right)^2 \tag{4}$$

where $v_{i/j}^{(*,e,m)}$ $(i < j)$ is the approximate truth computed based on the previous source weight W_i, and $v_j^{(\max,e,m)}$ is the absolute maximum value of $v_j^{(k,e,m)}(1 \leq k \leq K)$. We use $v_j^{(\max,e,m)}$ to normalize the distance between the optimal truth $v_{o,j}^{(*,e,m)}$ and the approximate one $v_{i/j}^{(*,e,m)}$ at t_j. Here, $v_{i/j}^{(*,e,m)}$ refers to $v_j^{(*,e,m)}$ in Definition 3.3. Specifically, let Φ represent Φ_i^{i-1}. The relationship between the unit error Φ and the source weight evolution is given by Theorem 1.

THEOREM 1. *Given a unit error threshold ε, let K be the size of source collection. If for all k, $1 \leq k \leq K$, the source weight evolution holds: $\Delta w_i^k \leq \sqrt{\varepsilon}/K$, then the unit error $\Phi \leq \varepsilon$ is satisfied.*

PROOF. According to Formulas (1) and (4), we derive the following:

$$\sqrt{\Phi} = \left| \frac{\sum_{k=1}^{K} (w_i^k / \sum_{k=1}^{K} w_i^k - w_{i-1}^k / \sum_{k=1}^{K} w_{i-1}^k) \cdot v_i^{(k,e,m)}}{v_i^{(\max,e,m)}} \right|$$

Then, we can infer

$$\sqrt{\Phi} \leq \sum_{k=1}^{K} \left| \frac{(w_i^k / \sum_{k=1}^{K} w_i^k - w_{i-1}^k / \sum_{k=1}^{K} w_{i-1}^k) \cdot v_i^{(k,e,m)}}{v_i^{(\max,e,m)}} \right|$$

Since $\left| v_i^{(\max,e,m)} \right| \geq \left| v_i^{(k,e,m)} \right|$ $(1 \leq k \leq K)$, we have

$$\sqrt{\Phi} \leq \sum_{k=1}^{K} \left| w_i^k / \sum_{k=1}^{K} w_i^k - w_{i-1}^k / \sum_{k=1}^{K} w_{i-1}^k \right|$$

Further,

$$\sqrt{\Phi} \leq K \cdot \sqrt{\varepsilon}/K = \sqrt{\varepsilon}$$

So far, we prove that $\Phi \leq \varepsilon$ holds. □

Theorem 1 demonstrates the relationship between the source weight evolution and the unit error, i.e., the unit error Φ should be no more than ε if the formula (5) is satisfied,

$$\Delta w_i^k \leq \sqrt{\varepsilon}/K \ (1 \leq k \leq K) \tag{5}$$

Under this scenario, we can use W_{i-1} to approximate W_i and ensure that the deviation between the optimal truth and the approximate one will be constrained by a threshold ε. Since we un-assess all sources weights at t_i, the time complexity of truth discovery is linear. For further improving the efficiency, we aim to assess source weights over time as few as possible. Therefore, it is essential to further analyze the relationship between the source weight evolution and the errors cumulated in a time period, i.e., the cumulative error, which is computed by Formula (6),

$$\Psi_j^i = \sum\nolimits_{h=i+1}^{j} \Phi_h^i \tag{6}$$

Combining with Formula (4), we can see that the cumulative error is defined as the sum of unit errors in a time period. Then, we give the maximum value of the cumulative error under the condition that Formula (5) holds in a time period.

THEOREM 2. *Given a unit error threshold ε, let K be the size of source collection. If for all k, h, $1 \leq k \leq K$, $i < h \leq j$, the source weight evolution holds: $\Delta w_h^k \leq \sqrt{\varepsilon}/K$, then the cumulative error Ψ_j^i meets the condition $\Psi_j^i \leq \Delta T(\Delta T + 1)(2\Delta T + 1)\varepsilon/6$, where $\Delta T = j - i$.*

PROOF. According to Formulas (1) and (4), we derive the following:

$$\sqrt{\Phi_h^i} = \left| \frac{\sum_{k=1}^{K} (w_h^k / \sum_{k=1}^{K} w_h^k - w_i^k / \sum_{k=1}^{K} w_i^k) \cdot v_h^{(k,e,m)}}{v_h^{(\max,e,m)}} \right|$$

Then similar to Theorem 1, we have

$$\sqrt{\Phi_h^i} \leq \sum\nolimits_{k=1}^{K} \left| w_h^k / \sum\nolimits_{k=1}^{K} w_h^k - w_i^k / \sum\nolimits_{k=1}^{K} w_i^k \right|$$

According to $\Delta w_h^k \leq \sqrt{\varepsilon}/K$, for any h $(i < h \leq j)$, it is easy to derive the following:

$$\sqrt{\Phi_h^i} \leq (h - i) \cdot \sqrt{\varepsilon}$$

Further,

$$\sum\nolimits_{h=i+1}^{j} \Phi_h^i \leq \sum\nolimits_{h=i+1}^{j} (h-i)^2 \varepsilon$$

Then,

$$\sum\nolimits_{h=i+1}^{j} \Phi_h^i \leq (j-i)(j-i+1)(2(j-i)+1)\varepsilon/6$$

Since $\Psi_j^i = \sum_{h=i+1}^{j} \Phi_h^i$, we have

$$\Psi_j^i \leq (j-i)(j-i+1)(2(j-i)+1)\varepsilon/6$$

Let $\Delta T = j - i$, we prove that $\Psi_j^i \leq \Delta T(\Delta T + 1)(2\Delta T + 1)\varepsilon/6$ holds. □

According to Theorem 2, we can get that the relationship between the unit error and the maximum value of cumulative error under the condition of $\Delta w_h^k \leq \sqrt{\varepsilon}/K \, (i < h \leq j, \ 1 \leq k \leq K)$:

$$\max(\Psi_j^i) = \Delta T(\Delta T + 1)(2\Delta T + 1)\varepsilon/6 \tag{7}$$

where $\Delta T = j - i$. Let the size of source collection K be 3 and the unit error threshold ε be 0.03. Suppose that we update the source weights at t_1 and the source weight evolutions satisfy Formula (5) from t_2 to t_5, i.e., $\Delta w_i^k \leq \frac{0.03}{3} = 0.01 \ (1 \leq k \leq 3, \ 1 < i \leq 5)$. The cumulative error Ψ_5^1 will be no more than $4 \times (4+1) \times (2 \times 4 + 1) \times 0.03/6 = 0.9$.

Theorem 2 ensures that, under dynamic scenario, we can incorporate iterative methods to improve the accuracy of truth discovery without scarifying much efficiency. The reason is that we neglect the iterative estimation of source weights W_i when the source weight evolutions Δw_i^k $(1 \leq k \leq K)$ satisfy Formula (5), i.e., the iterative truth discovery methods are utilized over data streams only at certain timestamps. In addition, as the cumulative error is constrained by ε and ΔT, we can ensure the accuracy of truth discovery even if the iterative process is reduced. Although we do not update the source weights at each timestamp, the accuracy of our method is still much higher than the existing incremental methods (as shown in Section 6).

To capture the temporal relations among truths by adding smoothing factor as in Formula (2), we only need to redefine $v_j^{(\max,e,m)}$ in Formula (4) as the absolute maximum value of $v_j^{(1,e,m)}, v_j^{(2,e,m)}, ..., v_j^{(K,e,m)}$, $v_{j-1}^{(*,e,m)}$, and slightly modify Formula (5) by changing K into $K+1$. The reason is that we treat the smoothing factor as the weight of the $(K+1)^{th}$ source and $v_{j-1}^{(*,e,m)}$ as the information from this source. Since we still compute truths by exploiting weighted combinations, the smoothing factor will not affect our conclusions. Moreover, we introduce the smoothing factor for truth computation only when the data changing is smooth, thus it is reasonable to utilize the $v_j^{(\max,e,m)}$ to normalize the unit error.

As shown in Theorems 1 and 2, a relative smooth source weight evolution leads to a lower unit error comparing with a big "jump" (the peaks in Figure 2) of source weight evolution. However, since the evolution of source weight is unknown over data streams, it is hard to make sure whether Formula (5) is satisfied. For solving this issue, we propose a probabilistic model to dynamically estimate the probability of Formula (5) holding over data streams.

5. ASRA-BASED TRUTH DISCOVERY

In this section, we propose an adaptive source reliability assessment scheme (ASRA) for truth discovery over data streams. The basic idea behind this scheme is to dynamically determine the time for source weight assessment. Then the truth with a predetermined accuracy is identified. Specifically, we first derive a probabilistic model to estimate the probability of the source weight evolution which meets the condition in Formula (5). By integrating the conclusions in section 4, we achieve the maximal period of source weight assessment under the condition that the maximum value of cumulative error is smaller than a given threshold in a certain confidence level. This will transform the source weights assessment into an optimization problem. Based on this optimization problem, we then propose our ASRA scheme that adaptively assesses source weights over streams.

5.1 Probability Forecasting Model

As proved in Theorem 1, the source weight evolution has great influence on unit error. If all the source weight evolutions meet the conditions in Formula (5), the unit error will be less than ε. Otherwise, it can not be controlled within the ε constraint. However, in real-world applications, even if the variation trend of the information from various sources can be obtained, the evolution of each source weight is still not available. Considering this, we propose a probability model based on the Bernoulli distribution to estimate the probability of Formula (5) holding over data streams. Given a timestamp t_i, we can consider $\Delta w_i^k \leq \sqrt{\varepsilon}/K$ $(1 \leq k \leq K)$ as an independent and random event. Here, the probability of the event occurrence is a random variable which follows Bernoulli distribution, i.e., $\xi \sim B(1,p)$. Based on the probability theory, the probability p can be estimated by sampling as explained by Example 1.

Example 1. *Given a unit error threshold ε and a source collection, assume that $t_1 \sim t_l$ is the initial period of time. We assess the source weight at each timestamp. Let N be the times of all source weight evolutions satisfying Formula (5) during this period. The total times of counting all source weight evolutions is $M = l - 1$. Thus the probability p can be estimated as N/M.*

As the time increases, both the source weight evolution and the probability p are likely to change. Thus, a dynamic estimation makes the probability p more accurate. This is also the basis of ASRA scheme. We will illustrate the time for the update of probability p while introducing our scheme.

5.2 ASRA Scheme

This section presents our ASRA scheme in details. The ASRA scheme includes two parts: (1) adaptive update point prediction; and (2) ASRA-based truth discovery algorithm. We first transform the update point prediction issue into an optimization problem which minimizes the frequency of source weight assessment. Then, an ASRA-based algorithm is proposed with the support of this optimization strategy. ASRA assesses source weights with changeable frequencies while finding the truth with a certain level of accuracy given by users. Accordingly, we can achieve high efficiency by reducing the frequency of assessing source weights and high accuracy by incorporating the iterative process. Given a current update point t_i, ASRA predicts the next update point t_j by solving the following optimization problem:

$$\begin{gathered} Max \quad j = i + \Delta T \\ s.t. \quad (\Delta T - 1)(\Delta T - 2)(2\Delta T - 3)\varepsilon/6 \leq E \\ p^{\Delta T - 2} \geq \alpha \end{gathered} \tag{8}$$

where ΔT is considered as the maximum period of assessing source weights. There are two constraint functions regarding this optimization problem as listed below:

- $p^{\Delta T-2} \geq \alpha$: This is equivalent to $p(\Delta w_h^k \leq \sqrt{\varepsilon}/K) \geq \alpha$ $(1 \leq k \leq K,\ i+1 < h < j)$, where α is the probability threshold given by users. We do not need to estimate the source weight evolutions at t_{i+1} and t_j. For one thing, we assess the source weights W_i since t_i is an update point. Considering that we should compute the source weight evolutions for dynamically updating p, the source weights W_{i+1} is also assessed to obtain the evolution of all source weights, i.e., $\Delta w_{i+1}^1, \ldots,\ \Delta w_{i+1}^K$. Then, we utilize W_{i+1} instead of $W_{i+2},\ \ldots,\ W_{j-1}$ to compute the truths at $t_{i+2},\ \ldots,\ t_{j-1}$. For another, we assess the source weights W_j since t_j is also an update point. Thus, it is unnecessary to estimate the probability of $\Delta w_{i+1}^k \leq \sqrt{\varepsilon}/K,\ \Delta w_j^k\ \leq \sqrt{\varepsilon}/K$ $(1 \leq k \leq K)$.
- $(\Delta T-1)(\Delta T-2)(2\Delta T-3)\varepsilon/6 \leq E$: Based on Theorem 2, when $p(\Delta w_h^k \leq \sqrt{\varepsilon}/K) \geq \alpha$ $(i+1 < h < j)$, the probability of $\max(\Psi_{j-1}^{i+1}) = (\Delta T-1)(\Delta T-2)(2\Delta T-3)\varepsilon/6$ is no smaller than α. Though we expect ΔT to be large for high efficiency, $\max(\Psi_{j-1}^{i+1})$ will become large with ΔT increasing. Thus, we also need to make sure that $\max(\Psi_{j-1}^{i+1})$ is no more than E, where E is the cumulative error threshold given by users. By this way, the cumulative error between any two update points is constrained.

Formula (8) implicates that our ASRA scheme tries to search for the maximum period of assessing source weights. When the unit error threshold ε is fixed, only two tuned parameters, α and E, need to be set. A large α may lead to a small ΔT, while a small E will also result in a small ΔT. However, the performance trend of ε is actually uncertain. We will show in Section 6 that the effects of the probability threshold α, cumulative threshold E and unit error threshold ε in our framework, and the performance of our framework can be flexibly changed by tuning these parameters.

Algorithm 1 presents the whole procedure of ASRA-based truth discovery. Let t_i denote the current timestamp and t_j denote the update point, Algorithm 1 performs in three steps. In the first step (lines 3-4), we update the source weights. Given the update points t_j and t_{j+1} (line 3), we call the existing truth discovery method to assess the source weights W_j, W_{j+1}. In the second step (lines 5-13), we update the probability p of satisfying Formula (5) by re-estimating p according to $\Delta w_{j+1}^k (1 \leq k \leq K)$. In the last step (lines 14-18), we predict the next update point. By utilizing the probability p computed in the second step, we predict the next update point t_j according to Formula (8). If ΔT computed by Formula (8) is less than 2, we set $\Delta T = 2$ (lines 16-17).

In Algorithm 1, line 4 suggests that various methods for source weight computation can be plugged into our scheme only if the truth computation of the plugged method is in the form of weighted combinations. We set a window size M for more accurately estimating the probability p without the influence of out-of-date data. Note that we can introduce the smoothing factor by slightly modifying our algorithm. As mentioned above, we treat the smoothing factor λ as the weight of $(K+1)^{th}$ source and the previous truths as the information from this source. As λ is a constant [11], only the source weight evolution and the size of source collection will be changed when the smoothing factor λ is introduced. Accordingly, for capturing the temporal relationship over streaming data, we only need to change K into $K+1$ in line 6, and change "Formula (1)" into "Formula (2)" in line 21. For the existing truth discovery methods plugged into our scheme (line 4), we also simply change its truth computation from "Formula (1)" into "Formula (2)". Obviously, the complexity of the algorithm is determined by the corresponding iterative truth discovery methods at an update point. Otherwise, its complexity is $O(|V_i|)$ at t_i.

For probability p, there are two points to remark: (1) the cumulative error is usually constrained to a small value in real world applications. According to Formula (8), ΔT will not be a large value. Thus we can assume that p is a constant in a small time window (ΔT); and (2) p is defined as the probability of all the source weight evolutions satisfying Formula (5) at each timestamp, i.e., a small p implies the source weight evolution is generally large over data streams.

```
Algorithm 1: ASRA-based truth discovery
Input  : Observation collection $V_i$, threshold $\alpha$, $E$;
Output: Truth collection $V_i^*$;
$j \leftarrow 1$, $m \leftarrow 1$, $N[1...M] \leftarrow 0$, $p \leftarrow 0$;
for $i = 1 \to \infty$ do
    if $i == j || i == j + 1$; then
        Update $V_i^*$, $W_i$ according to existing iterative
        truth discovery methods;
    if $i == j + 1$; then
        if all $\Delta w_i^k$ $(1 \leq k \leq K)$ satisfy Formula (5);
        then
            $N[m] = 1$;
        if $m <= M$; then
            $p = (\sum_{n=1}^{m} N[n])/m$
        else
            Slide the window forward and keep array $N$
            always contains $M$ elements;
            $p = (\sum_{n=1}^{M} N[n])/M$;
        $m++$;
        $i = i - 1$;
        Update $j$ by Formula (8);
        if $j - i < 2$; then
            $j = i + 2$;
        $i = i + 1$;
    else
        $W_i \leftarrow W_{i-1}$;
        Set $V_i^*$ by Formula (1);
Return $V_i^*$;
```

Note that the exact timestamp with a large source weight evolution is still unknown if we do not compute the source weights. Therefore, the algorithm may also neglect source weight computation when the source weight evolution does not satisfy Formula (5). However, according to Formula (8), a small p will lead to more frequent source weight estimation, thus the high performance of our framework can be ensured (as shown in Section 6).

6. EXPERIMENTS

In this section, we experimentally validate the proposed approach for truth discovery over data streams.

6.1 Experimental Setup

We evaluate our framework on three real-world datasets: Sensor Dataset[1], Stock Dataset[2] and Weather Dataset[2]. The Sensor Dataset contains data from 54 sensors deployed in the Intel Berkeley Research lab between Feb. 28, 2004 and Apr. 5, 2004. Each sensor collected the time-stamped topology values once per 30 seconds. The temperature and humidity properties are adopted for evaluation. The Stock Dataset contains data for 1000 stocks that are collected from 55 sources over the weekdays of July 2011. We adopt three properties: change %, change value and last trade price. The ground truths are given. The Weather Dataset contains 18 sources that record weather data for 30 cities of United States from Jan. 28, 2010 to Feb. 4, 2010. We adopt the temperature and humidity properties, and consider the information collected from Accuweather.com as the ground truths.

Since the ground truths of Stock Dataset and Weather Dataset are known, each source weight can be quantified by measuring the distance between its observations and the ground truths. Accordingly, **the true source weights of Stock Dataset and Weather Dataset are also available.** Moreover, although Stock Dataset and Weather Dataset have been used in [11], the experimental results can be different because we choose various types of properties to conduct the experiments while only one type of property was used in [11].

6.2 Evaluation Methodology

We have conducted extensive experiments to evaluate the effectiveness and efficiency of the proposed method by four steps: (1) validate the effectiveness of the probabilistic model estimateing source weight evolution; (2) analyze the effects of three parameters, probability threshold α, cumulative error threshold E and unit error threshold ε in our framework; (3) evaluate the effectiveness and efficiency of our approach by comparing with state-of-art competitors; and (4) further confirm the accuracy of source weight computation of our proposed approach. Eleven methods, including seven state-of-the-art competitors and four proposed alternatives, are used in the experiments.

Baseline Methods. The following state-of-the-art methods for truth discovery over continuous data are implemented. The parameters of each baseline method are set according to the original paper.

- GTM: Using Bayesian probabilistic model for resolving conflicts on continuous data [21].
- CRH: Working with heterogeneous data by incorporating into various loss functions [8].
- DynaTD: Finding truths over data streams in an incremental way [11].
- DynaTD+smoothing: Adding the smoothing factor based on DynaTD [11].
- DynaTD+decay: Adding the decay factor based on DynaTD [11].
- DynaTD+all: Adding both the smoothing factor and the decay factor based on DynaTD [11].
- Dy-OP: Optimization-based solution of DynaTD [11].

Proposed Alternatives. We plug different existing truth discovery methods into our framework. All these methods iteratively conduct the updates of source weights and truths until convergence. For the truth update, all these methods exploit weighted combinations strategy (i.e., Formula (1) or (2)) [8, 11] and can be plugged into our framework. The details on the source weight update for each method are as follows:

- ASRA(CRH): We incorporate CRH into our framework and choose the normalized squared loss function to measure the deviation from the truths to the observations. The source weight w_i^k is derived as the following formula:

$$w_i^k = -\log(\frac{l_i^k}{\sum_{k'=1}^{K} l_i^{k'}}) \tag{9}$$

[1] http://db.csail.mit.edu/labdata/labdata.html
[2] http://lunadong.com/fusionDataSets.htm

where l_i^k refers to the normalized squared loss function of the k^{th} source at t_i [8], i.e.,

$$l_i^k = \sum\nolimits_{e=1}^{E} \sum\nolimits_{m=1}^{M} \frac{\left(v_i^{(k,e,m)} - v_i^{(*,e,m)}\right)^2}{std(v_i^{(1,e,m)}, \dots, v_i^{(K,e,m)})} \qquad (10)$$

- ASRA(CRH+smoothing): We further introduce the smoothing factor λ to ASRA(CRH) for capturing the temporal relations over streams. Under this scenario, we consider $v_{i-1}^{(*,e,m)}$ as the information from the $(K+1)^{th}$ source ($v_{i-1}^{(*,e,m)} = v_i^{(K+1,e,m)}$) and λ is the weight of this source. Therefore, only the number of sources in Formula (10) and Formula (9) need to be changed for computing loss functions and source weights.

- ASRA(Dy-OP): We incorporate the basic optimization function of DynaTD [11], denoted as Dy-OP, into our framework. The source weight w_i^k is derived as the following formula:

$$w_i^k = \frac{q_i^k}{\eta \cdot l_i^k} \qquad (11)$$

  where q_i^k refers to the number of observations provided by the k^{th} source at t_i and η is a trade-off parameter of Dy-OP [11]. In addition, the normalized squared loss functions l_i^k $(1 \le k \le K)$ in Formula (11) are computed by Formula (10).

- ASRA(Dy-OP+smoothing): The smoothing factor λ is also introduced to ASRA(Dy-OP) for capturing the temporal relations over streaming data. As mentioned, only the number of sources need to be changed for computing source weights and loss functions.

So far, for each method plugged into our framework, we have presented the formula for its source update step. The details of Formulas (9) and (11) are listed in Appendix. For truth computation, we only need to utilize Formula (2) to capture the temporal relations over streaming data.

Performance Metrics. To evaluate the efficiency of our framework, we report the *running time* of each method. To assess the accuracy of it, we calculate the *Mean of Absolute Error* (MAE) of each method by comparing their outputs with ground truths. For both metrics, lower values indicate better performance. All the algorithms were performed on a PC with Windows OS, Intel Core i7 processor.

6.3 Probabilistic Model Validation

This part validates the effectiveness of the probabilistic model for estimating the source weight evolution over data streams. Obviously, if the probabilistic model can capture the large source weight evolution (Formula (5) cannot be satisfied), our proposed model is effective. Thus, we validate the effectiveness of our probabilistic model by counting all probable scenarios including:

(1) Formula (5) does not hold and our framework updates the source weights at the same time (denoted as TP);

(2) Formula (5) holds and our framework keeps the source weights at the same time (denoted as TN);

(3) Formula (5) does not hold and our framework keeps the source weights at the same time (denoted as FN);

Table 2: Probabilistic Model Valiadation

(a) Stock Dataset

Parameter Setting		Experimental Results				
ε	α	TP	TN	FN	FP	CR
5×10^{-4}	0.45	0.500	0.278	0.167	0.055	0.778
1×10^{-3}	0.45	0.390	0.333	0.222	0.055	0.723
5×10^{-3}	0.45	0.155	0.500	0.112	0.233	0.655
5×10^{-4}	0.55	0.500	0.278	0.167	0.055	0.778
1×10^{-3}	0.55	0.500	0.389	0.056	0.055	0.889
5×10^{-3}	0.55	0.212	0.444	0.055	0.289	0.656
5×10^{-4}	0.65	0.612	0.278	0.055	0.055	0.890
1×10^{-3}	0.65	0.612	0.333	0	0.055	0.945
5×10^{-3}	0.65	0.389	0.444	0.055	0.112	0.833

(b) Weather Dataset

Parameter Setting		Experimental Results				
ε	α	TP	TN	FN	FP	CR
5×10^{-2}	0.45	0.155	0.540	0	0.305	0.695
1×10^{-1}	0.45	0.058	0.724	0.023	0.195	0.782
5×10^{-1}	0.45	0.034	0.799	0	0.167	0.833
5×10^{-2}	0.55	0.155	0.495	0	0.350	0.650
1×10^{-1}	0.55	0.052	0.695	0.029	0.224	0.747
5×10^{-1}	0.55	0.034	0.776	0	0.190	0.810
5×10^{-2}	0.65	0.255	0.431	0.006	0.308	0.686
1×10^{-1}	0.65	0.063	0.632	0.017	0.288	0.695
5×10^{-1}	0.65	0.035	0.747	0	0.218	0.782

(4) Formula (5) holds and our framework updates the source weights at the same time (denoted as FP).

Both scenario (1) and (2) show that our probabilistic model captures the source weight evolution successfully. Thus, the effectiveness of our probabilistic model can be transformed into *Capture Rate* (CR) formulated as:

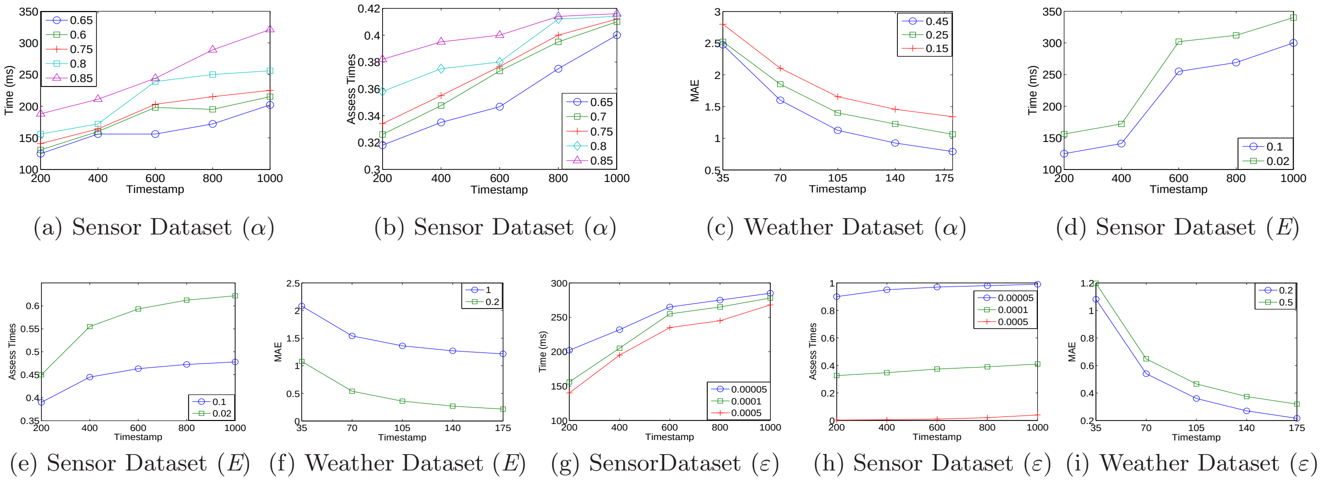
$$CR = TN + TP \qquad (12)$$

The experiments are conducted over Stock Dataset and Weather Dataset. We vary two parameters, α and ε, to observe the effectiveness of our probabilistic model with different parameter settings. The cumulative threshold E is given to constrain the maximum of ΔT.

The experimental results are reported in Table 2. As we can see, CR is always more than 0.6 on both two datasets and can achieve more than 0.9 at some cases. Note that our framework assigns the first two timestamps to update points, which may lead to a higher FP and a lower CR. Therefore, our probabilistic model can capture the source weight evolution in most situations, which further proves the effectiveness of our framework.

6.4 Evaluation on Parameters

To analyze the effects of the probability threshold α, cumulative error threshold E and unit error threshold ε in our framework, we test the performance of our method over the Sensor Dataset and Weather Dataset by changing the value of one parameter while fixing the others. To discover truths, we incorporate Dy-OP into our framework, i.e., ASRA(Dy-OP). Three metrics, running time, MAE and assess times, are used to observe the influence of three parameters to our framework. Here, *assess times* is defined as the average times of assessing source weight over streaming data. Ob-



(a) Sensor Dataset (α) (b) Sensor Dataset (α) (c) Weather Dataset (α) (d) Sensor Dataset (E)

(e) Sensor Dataset (E) (f) Weather Dataset (E) (g) SensorDataset (ε) (h) Sensor Dataset (ε) (i) Weather Dataset (ε)

Figure 3: Evaluation on Parameters

viously, lower assess times indicates higher efficiency and lower accuracy.

6.4.1 Effect of α

In this test, we evaluate the effect of the probability threshold α on the accuracy and efficiency of our framework. For Sensor Dataset, we fix ε to 10^{-5} and E to 1, and vary the value of α from 0.65 to 0.85. For Weather Dataset, we fix ε to 0.1 and E to 1, and vary the value of α from 0.15 to 0.45. The results are shown in Figures 3(a)-(c).

As we can see, with the increasing of α, running time and assess times increase while MAE decreases. This result is caused by the following reason. The probability threshold α controls the holding probability of Formula (5) during the period of keeping source weights. Therefore, a relatively large α means Formula (5) should be more likely hold, and a smaller α will relax this constraint while leading to a relatively large ΔT. In other words, a lager α achieves a higher accuracy while suffering from much sacrifice on efficiency.

6.4.2 Effect of E

In this test, we evaluate the effect of the cumulative error threshold E on the performance of our framework. For Sensor Dataset, we set E to 0.02 and 0.1 respectively, and fix ε to 10^{-5} and α to 0.75. For Weather Dataset, we set E to 0.2 and 1 respectively, and fix ε to 0.1 and α to 0.2. The results are shown in Figures 3(d)-(f).

Obviously, with the decreasing of cumulative threshold E, running time and assess times increase while MAE decreases. According to Formula (8), a relatively large E means our framework is allowed to make more errors between any two update points. Therefore, a large E will lead to a large period of assessing source weights and improve the efficiency. However, it suffers from much sacrifice on accuracy.

6.4.3 Effect of ε

We test the effect of the unit error threshold, ε, on three metrics. For Sensor Dataset, we fix α to 0.6 and E to 0.01, and set ε to 5×10^{-5}, 10^{-4} and 5×10^{-4} respectively. For Weather Dataset, we fix α to 0.95 and E to 1, and set ε to 0.2 and 0.5 respectively. The results are shown in Figures 3(g)-(i).

As we can observe, with the increasing of ε, running time and assess times decrease while MAE increases. However, the performance trend of unit error threshold is actually uncertain. Based on the first constraint function of Formula (8), a relatively small ε may result in a larger ΔT. At the same time, the second constraint of Formula (8) implicates that a relatively small ε can also result in a smaller ΔT. Since we set a relatively large cumulative error threshold E ($E = 1$) in our experiments, the optimal ΔT is mainly restricted by the second constraint function of Formula (8). Thus a larger ε achieves a better efficiency and suffers from much sacrifice on accuracy.

For the same parameter setting, with the time increasing, MAE decreases while both running time and assess time increase over two datasets. This is because the source weight evolutions of these two datasets become large as the time increases. Thus our framework automatically improve the frequency of assessing source weights and achieve the high accuracy of the truth discovery.

To summarize, all the experimental results (Figures 3(a)-(i)) show that these three parameters of our framework can tune the performance of truth discovery flexibly.

6.5 Evaluation on Performance

We first compare our proposed approach with the state-of-the-art competitors in terms of effectiveness and efficiency. Then, we further study the effectiveness of our approach under the optimal efficiency, and its efficiency under the best accuracy.

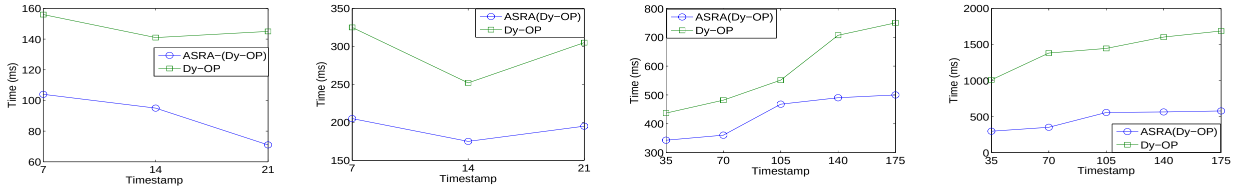
6.5.1 Comparison with Existing Approaches

In this test, we evaluate our proposed approach by comparing with the existing competitors: DynaTD, DynaTD+smoothing, DynaTD+decay, DynaTD+all, Dy-OP, CRH and GTM. For Stock Dataset, we set ε to 10^{-3}, α to 0.75 and E to 1. For Weather Dataset, we set ε to 0.1, α to 0.8 and E to 1. For Sensor Dataset, we set ε to 5×10^{-6}, α to 0.85 and E to 0.01. Table 3 shows the experimental results for all the methods on the three datasets. Since the ground truths of Sensor Dataset are unknown, we only report the accuracy (MAE) on two datasets with ground truths, i.e., Stock Dataset and Weather Dataset.

Efficiency. In terms of efficiency, the proposed method performs nearly as well as DynaTD, DynaTD+smoothing,

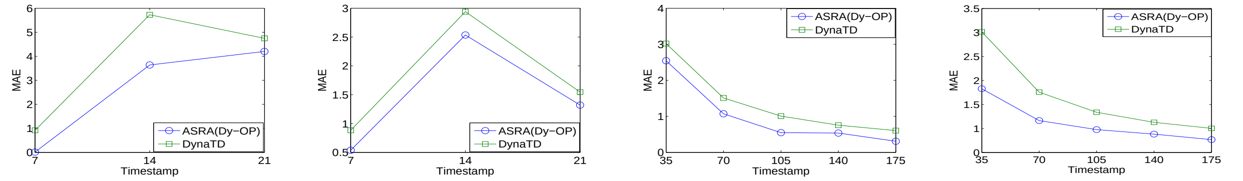
Table 3: Comparison with Existing Approaches

Method	Stock Dataset		Weather Dataset		Sensor Dataset
	MAE	Time(ms)	MAE	Time(ms)	Time(ms)
ASRA(Dy-OP)	1.3941	99	0.4974	419	658
ASRA(CRH)	1.4007	104	0.5029	424	674
ASRA(Dy-OP+smoothing)	1.0142	103	0.4474	417	638
ASRA(CRH+smoothing)	1.0781	117	0.5076	427	676
DynaTD	1.5462	99	1.0593	316	549
DynaTD+smoothing	1.5064	98	0.9261	306	595
DynaTD+decay	1.4956	98	0.9300	310	552
DynaTD+all	1.4455	93	0.9205	307	570
Dy-OP	1.3328	305	0.4425	1680	2041
CRH	1.3994	325	0.5028	1782	2092
GTM	1.4112	430	0.6011	1718	2133



(a) Stock Dataset (Sin) (b) Stock Dataset (Mul) (c) Weather Dataset (Sin) (d) Weather Dataset (Mul)

Figure 4: Efficiency Study



(a) Stock Dataset (Sin) (b) Stock Dataset (Mul) (c) Weather Dataset (Sin) (d) Weather Dataset (Mul)

Figure 5: Accuracy Study

DynaTD +decay and DynaTD+all. As all these methods work in an incremental way, they can be viewed as the low bound of the iterative methods. Therefore, the results shown in Table 3 implicate our proposed framework can achieve high efficiency. Meanwhile, ASRA(Dy-OP) can run as fast as DynaTD on Stock Dataset. The reason is that the proposed framework only performs iterations at certain timestamps. Moreover, our proposed framework is more efficient compared with other iteration-based truth discovery methods. Specifically, our framework outperforms the iterative method GTM in terms of both accuracy and efficiency. The reason is that the basic methods plugged into our framework (CRH, Dy-OP) achieve better performance than GTM.

Accuracy. In terms of effectiveness, the proposed method is better than existing competitors, DynaTD, DynaTD+smoothing, DynaTD+decay and DynaTD+all. The reason is that these competitors exploit incremental computation, updating the source weights according to the new arrival data until each source weight converges to a certain value. However, the true source weights in real applications are constantly changing. Thus, the source weights computed by the incremental methods deviate from the true ones, leading to big errors. In addition, CRH and Dy-OP are more accurate than our methods (ASRA(CRH), ASRA(Dy-OP)), as they solve the truth discovery task by an iterative process that iteratively computes the truths and source weights at each timestamp. In this way, each source weight converges to its optimal one. However, without computing the source weights at each timestamp, the accuracy of ASRA(Dy-OP) and that of ASRA(CRH) are still similar to the corresponding basic methods Dy-OP and CRH. The reason is that our proposed framework updates the source weights frequently when the source weight evolutions are generally large. Based on Theorems 1 and 2, we can constrain the cumulative error and ensure the accuracy of our framework. When a smoothing factor is introduced, our methods, ASRA(Dy-OP+smoothing) and ASRA(CRH+smoothing), achieve the best accuracy among all the methods on Stock Dataset. It can also be observed that ASRA(Dy-OP) achieves better accuracy than ASRA(CRH), while Dy-OP performs better than CRH on both two datasets. Obviously, the accuracy of our framework is consistent with the basic method plugged into it.

In conclusion, from the performance comparison results, it can be seen that our framework always outperforms the iterative methods with respect to efficiency and performs better than the incremental methods in terms of accuracy.

Since our framework can contain different plugged truth discovery methods, it also outperforms some baselines in terms of both accuracy and efficiency (such as GTM).

6.5.2 Further Study

To further confirm the performance of our framework, we evaluate its efficiency while achieving the optimal accuracy, and its accuracy while the efficiency is optimal. In this test, we conduct experiments on Stock Dataset and Weather Dataset. Since our framework can flexibly tune the efficiency and accuracy of truth discovery over streaming data, both accuracy and efficiency can be optimized by tuning the parameters. Also, we change the number of properties in this part, and denote the experiments conducted on a single property as Single-Property ("Sin" in Figures (4)-(5)), and the ones on multiple properties as Multiple-Property ("Mul" in Figures (4)-(5)). For evaluation on Single-Property, we choose the last trade price property for Stock Dataset, and the humidity property for Weather Dataset.

Efficiency. From Table 3, we can see that Dy-OP achieves the best accuracy comparing with all the baselines. Thus, the accuracy of Dy-OP can be considered as the optimal accuracy. We achieve the same accuracy with Dy-OP by tuning the parameters ($\varepsilon = 10^{-3}$, $\alpha = 0.85$, $E = 0.1$ for Stock Dataset and $\varepsilon = 10^{-3}$, $\alpha = 0.85$, $E = 1$ for Weather Dataset). Under this scenario, we evaluate the efficiency of our framework by comparing with Dy-OP.

From Figures 4(a)-(d), we can see that our framework achieves much higher efficiency performance than Dy-OP for both Single-Property and Multi-Property. The reason is that our framework does not assess the source weights continually. In addition, the gap between our framework and Dy-OP on Multiple-Property is larger than the one on Single-Property, which illustrates our method is more suitable for addressing different types of properties.

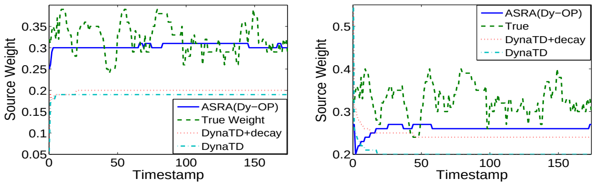
Accuracy. To the best of our knowledge, DynaTD is the most effective incremental truth discovery method for continuous data, and also the basis of DynaTD+smoothing, DynaTD+decay, DynaTD+all [11]. Thus, we consider the efficiency of DynaTD as the optimal efficiency. Then we achieve the same efficiency with DynaTD by tuning the parameters ($\varepsilon = 10^{-3}$, $\alpha = 0.75$, $E = 1$ for Stock Dataset and $\varepsilon = 0.1$, $\alpha = 0.65$, $E = 1$ for Weather Dataset). Under this scenario, we evaluate the accuracy of our framework by comparing with DynaTD.

Figures 5(a)-(d) show that, for both Single-Property and Multi-Property, the accuracy of our proposed framework is much higher than the incremental method. For one thing, we use the iterative method to assess source weights, which makes source weights converge to the optimal values at each timestamp. For another, both Theorems 1 and 2 ensure the accuracy of our framework. Although we do not assess the source weights continually, our framework achieves much higher accuracy comparing with the existing incremental methods. Moreover, Figure 5(a) shows that, at the initial time, the truths computed by our framework is nearly equal to the ground truths, which also implicates the high accuracy of our framework.

To summarize, by tuning the parameters of our framework, we can balance the efficiency and accuracy of the truth discovery task, and achieve better performance than the state-of-the-art competitors as well.

6.6 Evaluation on Source Weight

As aforementioned, the estimation of source weights plays a vital role in the truth discovery task. Thus, we design a set of experiments to evaluate the accuracy of source weight computation using our proposed framework. In this test, we choose Weather Dataset as the experimental dataset. We randomly select two sources (denoted as S_1, S_2 respectively) for experiments. Dy-OP method is plugged into our framework, i.e., ASRA(Dy-OP). For comparison purpose, we also compute the source weights using the existing incremental methods, DynaTD and DynaTD+decay. Moreover, for controlling the source weights in a same range, we utilize L^1-norm to regularize the source weights computed by all the methods.



(a) Weather Dataset (S_1) (b) Weather Dataset (S_2)

Figure 6: Evaluation on Source Weight

Figures 6(a)-(b) show the experimental results. Clearly, each true source weight changes constantly over time, and the source weights computed by our framework are usually more closer to the true values. Conversely, a source weight computed by DynaTD and DynaTD+decay can converge to a certain value quickly, which is inconsistent with the real source weight change. In conclusion, these results prove the accuracy of our approach in terms of source weight computation.

7. CONCLUSION

In this paper, we study the truth discovery problem over data streams. We propose a framework for truth discovery which adaptively determines the frequency of assessing source weights for high efficiency and incorporates various iterative truth discovery methods for high accuracy. We first define and study the unit error and the cumulative error of truth discovery. Then we transform the prediction of the cumulative error into an optimization problem, and propose our ASRA scheme. Tuning parameters of our framework supports a trade-off between accuracy and efficiency in truth discovery. Moreover, by a series of theoretical analysis, the accuracy of our framework is guaranteed while the iterative processes are reduced. Extensive experiments on real-world datasets have been conducted to evaluate the effectiveness and efficiency of our approach, and the experimental results have proved the high performance of our truth discovery framework.

8. ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (61472071 and 61433008) and the National Basic Research Program of China (973 Program) under Grant No. 2012CB316201. Yu Gu is the corresponding author.

9. REFERENCES

[1] X. L. Dong, L. BertiEquille, Y. Hu, and D. Srivastava. Global detection of complex copying relationships between sources. *PVLDB*, 3(12):1358–1369, 2010.

[2] X. L. Dong, L. BertiEquille, and D. Srivastava. Integrating conflicting data: the role of source dependence. *PVLDB*, 2(1):550–561, 2009.

[3] X. L. Dong, L. BertiEquille, and D. Srivastava. Truth discovery and copying detection in a dynamic world. *PVLDB*, 2(1):562–573, 2009.

[4] X. L. 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 *SIGKDD Conference Proceedings*, pages 601–610, 2014.

[5] X. L. Dong, E. Gabrilovich, K. Murphy, and V. Dang. Knowledge-based trust: Estimating the trustworthiness of web sources. *PVLDB*, 8(9):938–949, 2015.

[6] A. Galland, S. Abiteboul, A. Marian, and P. Senellart. Corroborating information from disagreeing views. In *WSDM Conference Proceedings*, pages 131–140, 2010.

[7] Q. Li, Y. Li, J. Gao, M. Demirbas, B. Zhao, L. Su, W. Fan, and J. Han. A confidence-aware approach for truth discovery on longtail data. *PVLDB*, 8(4):425–436, 2014.

[8] 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 Conference Proceedings*, pages 1187–1198, 2014.

[9] X. Li, X. L. Dong, K. Lyons, W. Meng, and D. Srivastava. Truth finding on the deep web: is the problem solved? *PVLDB*, 6(2):97–108, 2012.

[10] Y. Li, J. Gao, C. Meng, Q. Li, L. Su, B. Zhao, W. Fan, , and J. Han. A survey on truth discovery. *SIGKDD Explorations*, 17(2):1–16, 2015.

[11] Y. Li, Q. Li, J. Gao, L. Su, W. Fan, and J. Han. On the discovery of evolving truth. In *SIGKDD Conference Proceedings*, pages 675–684, 2015.

[12] F. Ma, Y. Li, Q. Li, M. Qiu, J. Gao, S. Zhi, L. Su, B. Zhao, H. Ji, and J. Han. Faitcrowd: Fine grained truth discovery for crowdsourced data aggregation. In *SIGKDD Conference Proceedings*, pages 745–754, 2015.

[13] S. Mukherjee, G. Weikum, and C. Danescu-Niculescu-Mizil. People on drugs: Credibility of user statements in health communities. In *SIGKDD Conference Proceedings*, pages 65–74, 2014.

[14] J. Pasternack and D. Roth. Latent credibility analysis. In *WWW Conference Proceedings*, pages 1009–1020, 2013.

[15] R. Pochampally, A. D. Sarma, X. L. Dong, A. Meliou, and D. Srivastava. Fusing data with correlations. In *SIGMOD Conference Proceedings*, pages 433–444, 2014.

[16] T. Rekatsinas, X. L. Dong, and D. Srivastava. Characterizing and selecting fresh data sources. In *SIGMOD Conference Proceedings*, pages 919–930, 2014.

[17] D. Wang, L. Kaplan, H. Le, and T. Abdelzaher. On truth discovery in social sensing: A maximum likelihood estimation approach. In *ISPN Conference Proceedings*, pages 233–244, 2012.

[18] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding. Data mining with big data. *IEEE Trans. Knowl. Data Eng.*, 26(1):97–107, 2014.

[19] X. Yin, J. Han, and P. S. Yu. Truth discovery with multiple conflicting information providers on the web. *IEEE Trans. Knowl. Data Eng.*, 20(6):796–808, 2007.

[20] L. Yu, J. Li, S. Cheng, S. Xiong, and H. Shen. Secure continuous aggregation in wireless sensor networks. *IEEE Trans. Parallel Distrib. Syst.*, 25(3):762–774, 2014.

[21] B. Zhao and J. Han. A probabilistic model for estimating real-valued truth from conflicting sources. In *QDB Conference Proceedings*, 2012.

[22] B. Zhao, B. I. P. Rubinstein, J. Gemmell, and J. Han. A bayesian approach to discovering truth from conflicting sources for data integration. *PVLDB*, 5(6):550–561, 2012.

[23] Z. Zhao, J. Cheng, and W. NG. Truth discovery in data streams: A single-pass probabilistic approach. In *CIKM Conference Proceedings*, pages 1589–1598, 2014.

[24] S. Zhi, B. Zhao, W. Tong, J. Gao, D. Yux, H. Jix, and J. Han. Modeling truth existence in truth discovery. In *SIGKDD Conference Proceedings*, pages 1543–1552, 2015.

APPENDIX

A. PROOF OF FORMULA (9)

PROOF. According to [8], for each timestamp t_i, the source weights W_i are conducted as the following:

$$W_i \leftarrow \arg\min_{W_i} \sum_{k=1}^{K} w_i^k l_i^k \quad s.t. \quad \sum_{k=1}^{K} \exp(-w_i^k) = 1 \tag{13}$$

Then the derivation of Formula (9) is the same as the derivation of source weights in [8].

□

B. PROOF OF FORMULA (11)

PROOF. According to [11], as we model that each source weight changes over time, the source weights W_i can be conducted as the following:

$$W_i \leftarrow \arg\min_{W_i} \eta \sum_{k=1}^{K} w_i^k l_i^k - \sum_{k=1}^{K} q_i^k \log(w_i^k) \tag{14}$$

where q_i^k denotes the number of observations provided by k^{th} source at t_i, and η is given to support the trade-off between the two terms in Formula (14) [11]. Moreover, the initial loss function in [11] is un-normalized. However, in this paper, we choose the normalized squared loss function for addressing different types of attributes (Formula (10)). Since the standard deviation of the observations at each timestamp can be considered as a constant, the conclusions will not be affected. We take the partial derivative of W_i in Formula (14) with respect to w_i^k, and set the partial derivative equal to zero. Then we obtain the source weight expression as shown in Formula (11).

□

Maritime Data Integration and Analysis: Recent Progress and Research Challenges

Christophe Claramunt, Cyril Ray, Loïc Salmon
Naval Academy Res. Inst.
Lanvéoc, France

Elena Camossi, Melita Hadzagic, Anne-Laure Jousselme
CMRE, La Spezia, Italy

Gennady Andrienko, Natalia Andrienko
Fraunhofer Institute IAIS
Sankt Augustin, Germany

Yannis Theodoridis, George A. Vouros
University of Piraeus
Piraeus, Greece

ABSTRACT

The correlated exploitation of heterogeneous data sources offering very large historical as well as streaming data is important to increasing the accuracy of computations when analysing and predicting future states of moving entities. This is particularly critical in the maritime domain, where online tracking, early recognition of events, and real-time forecast of anticipated trajectories of vessels are crucial to safety and operations at sea. The objective of this paper is to review current research challenges and trends tied to the integration, management, analysis, and visualization of objects moving at sea as well as a few suggestions for a successful development of maritime forecasting and decision-support systems.

Keywords

Big Spatio-temporal Data, Moving Objects, Maritime Information Systems, Event detection, Forecasting, Uncertainty

1. INTRODUCTION

The maritime environment has a huge impact on the global economy and our everyday lives. Specifically, Maritime Situation Awareness (MSA) and surveillance systems have been attracting increasing attention due to their importance for the safety and efficiency of maritime operations. Safety and security are constant concerns of maritime navigation, especially when considering the continuous growth of maritime traffic around the world and persistent decrease of crews on-board. 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). This has favoured and led to the development of automated monitoring systems, such as the Automatic Information Systems (AIS) and institutional initiatives for maritime data infrastructures [31]. However, the necessary correlated exploitation of large data sources offering historical and streaming maritime data is still a crucial computational issue. For instance, a typical volume of radio and satellite-based worldwide maritime data represents an estimated 18 millions positions per day [16] (see Figure 1 for an illustration of AIS coverage at the global level).

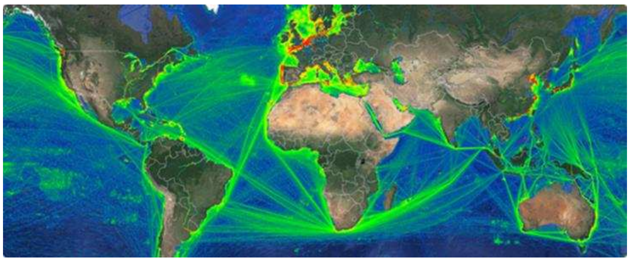


Figure 1: Worldwide AIS positions acquired by satellites (ORBCOMM)

Beside the indisputable value of information extracted from the AIS, the correlated exploitation of additional and heterogeneous sources is unavoidable to overcome the lack of veracity and incompleteness of the data. Thus, additionally to volume, velocity and variety, veracity of maritime data poses significant challenges. In particular, AIS messages are vulnerable to manipulation and subject to hacking [44], due to the unsecured channel of transmission, which weakens the whole system and the safety of navigation [35]. AIS data can thus contain deliberate falsifications and undergo spoofing [36], such as identity fraud, obscured destinations, or GPS manipulations [43]. According to [44], approximately 50% of AIS static data transmissions have errors of any kind. Vessels involved in illicit activities such as illegal fishing, deliberately avoid transmitting their information, while others may simply want to keep secret their fishing area to others.

Moreover, despite large available volumes, AIS data at open seas or at the border of Exclusive Economic Zones (EEZs) may be sparse, or delayed due to either low coverage or to multi-level processing issues. The data sparseness, latency, possibly manipulated, and the poor quality of movement data in general [1], render very challenging the design of information systems to support MSA processing AIS data and detecting abnormal behaviours. Thus, in addition to the need of real-time processing of large volume of data of high velocity, maritime surveillance systems should also have the ability to process and correlate many data sources, ideally of wide variety to compensate any lack of veracity of the data. For instance, Long Range Identification and Tracking (LRIT) and Vessel Monitoring Systems (VMS), Synthetic aperture radar (SAR) imagery can be used to verify AIS emission and detect anomalies [19].

Indeed, vessel trajectories are quite unique with respect to terrestrial trajectories: they are in only a limited way constrained by rigid network infrastructures, landmarks (e.g., ports), prefixed waiting / meeting points, and more difficult to observe and monitor. The objective of this paper is to review recent development and research challenges of methods, operational frameworks and systems oriented at large towards moving objects at sea. Most of the ideas and proposals developed in this paper are generated by the datAcron European funded project, which aims to advance the management and integrated exploitation of voluminous and maritime data sources, so as to significantly advance the capacities of systems to promote safety and effectiveness of critical operations for large numbers of moving entities in large geographical areas [12].

Our perspective towards an integrated maritime information infractructure is presented in Figure 2. The different components identified cover the integration of in-situ streaming data, trajectories detection and forecasting, recognition and identification of complex events and the development of visual analytics interfaces for maritime experts and decision-makers.

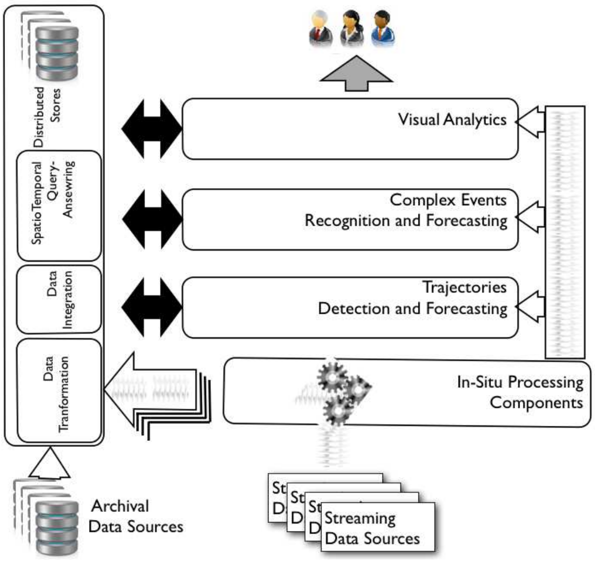

Distributed Stores
SpatioTemporal Query-Answering
Data Integration
Data Tranformation
Visual Analytics
Complex Events Recognition and Forecasting
Trajectories Detection and Forecasting
In-Situ Processing Components
Archival Data Sources
Streaming Data Sources

Figure 2: Towards an integrated maritime information infrastructure [12]

The rest of the paper is organised as follows. Section 2 introduces the main issues and challenges behind the integration of very large and heterogeneous maritime data and briefly surveys recent progress in information fusion, in-situ processing and database integration. Section 3 presents recent progress and remaining directions to explore in maritime event pattern and abnormal behaviour detection, trajectory analysis and visualisation. Section 4 explores issues, challenges and trends in maritime decision support and forecasting. Finally, Section 5 draws the conclusions.

2. MARITIME DATA INTEGRATION AND MANAGEMENT

The search for successful MSAs implies multiple data sources, such as surveillance sensors, automated processors, maritime institutional databases (e.g., navigation rules, protected areas), ocean and weather data, and "soft" data in unstructured formats (e.g., social media, intelligence reports). Nonetheless, advances in the spatio-temporal data analytics field with application on the urban domain (e.g., the very recent work over NYC Urban collection [10]) are not easily applicable in the maritime case. Efficient integration and management of maritime data is instrumental in effectively exploiting the available data, but it nevertheless entails some challenges that will be discussed in this section.

2.1 In-situ data processing

To face challenges due to the volume, velocity and variety of data sources, in-situ processing aims to scale, by shortening the time needed for detecting patterns of interest within a single- or cross-streaming process; addressing this challenge has been the focus of a great deal of academic research and industry efforts in recent years [11]. For instance, a framework for a distributed stream processing architecture supporting in-situ processing has been presented in [5]. However, such approaches have to become communication efficient and have to learn abilities for automatic model adaption for handling concept drift.

In-situ processing for the detection of patterns can be worthly-investigated towards cross-streaming data integration, and integration of streaming data (e.g., regarding a specific vessel) with contextual information (e.g., weather data) given that detected patterns may further be joined and aggregated, producing output streams that provide semantically and contextually rich information, further enabling effectiveness in detection and predictive analytics.

Closely related to the in-situ processing paradigm is the computation of data synopses. In particular, the computation of trajectories synopses for individual vessels is challenging, given that state of the art techniques [29] have achieved a compression ratio of 95% over AIS vessel traces. The challenge here is to address high levels of data compression without compromising the accuracy of the prediction / detection components.

2.2 Streaming data integration

In close relation to the computation of data synopses, a major objective is to develop appropriate components for integrating and summarizing maritime streaming data sources producing a scalable framework for cross-streaming data integration. Summarized streams can be semantically integrated with archival data as well as with detected and forecasted vessel trajectories and events.

The database community has proposed many efficient algorithms for link discovery applied to RDF data [32] [39]. Major shortcomings of these approaches are (a) the restriction to RDF properties of specific (mostly numerical) types, (b) the not-proved ability to integrate in real-time (cross-) streaming with archival data. Viewing the annotation of trajectories with contextual data as part of link discovery task, a specific challenge is the computation of semantic trajectories, taking advantage of trajectory-specific works (e.g., [34]). Distributed paradigms for streaming data, such as Storm, Spark Streaming, and Flink Streaming provide richer sets of primitive for incremental data sources, however, they do not provide full support for the integration of heterogeneous and historical data sources as well as for semantic enrichment and querying facilities [21] [37]. Moreover, these systems still lack specific spatio-temporal primitives necessary to deal with moving object trajectory data management [15].

2.3 Streaming data management

Recent works focus on the volume of spatial data to be processed, developing systems specifically oriented to the spatial domain and particularly moving objects [29] [38]. Other works have been extended to process data "on the fly" to handle data velocity and provide fast response time in a "moving object context" [34]. Nevertheless, these systems are oriented either towards a "posteriori analysis" characterized by long processing times or "on the fly processing" which can provide approximate answers to queries.

Provision of integrated views of data for exploring and querying streaming and archival data sources in real-time has also been addressed in the context of the Internet of Things (IoT): Live Knowledge Graphs, backed with scalable and elastic software stack can deal with millions of static records and billions of streaming triples per hour in real time [22] [26]. However, support of spatio-temporal queries and real-time integration of disparate data sources enabling scalability for massive amounts of dynamic data still remains a challenge. Concerning the representation and querying of spatial information, several RDF stores have begun integrating spatial query processing and reasoning [27]. Their performance still falls largely behind standard spatially-enabled DBMS's. Parallel and distributed platforms [46], key-value stores [33] and main memory systems, such as TriAD [20] and Trinity [47] have been developed for RDF data. However, current RDF stores with spatial and/or temporal support are not tailored to offer efficient trajectory-oriented data management, due to the volatile, multi-dimensional, and inherently sequential nature of such data (e.g., Strabon [24]).

2.4 Maritime data fusion

The information fusion literature addresses extensively the integration and combination of information from cooperative and non-cooperative maritime data sources. Information (data) fusion originally focused on "low-level" processing mainly from signals or images, from which vessel tracks (and trajectories) are built, new sensor measurements (contacts) are associated to tracks, objects corresponding to tracks are recognised and identified. The corresponding challenges include alignment of data in space and time, multi-resolution issues, at the same time, handling contextual and semantic differences.

The more recent trend in information fusion is higher-level processing with tasks, such as situation and impact assessments, closer to the decision maker, where the semantics has a higher importance, and involving human sources (hard and soft fusion) [13]. The integration and fusion of maritime data and information from various sources can overcome some of the single source processing issues (e.g., compensating for the lack of coverage and increasing accuracy). However, this also requires a suitable management of conflicting information, which may be either due to unintentional malfunction of sensors or to deliberate deception deemed of interest. The cross-fertilisation of database and fusion techniques will contribute to MSA just like the InFuse framework [14] or, more recently, the architecture proposed in [18].

2.5 Maritime data semantics and ontologies

A few semantic approaches, including vocabularies, taxonomies and ontologies have been proposed as tentatives to bridge the gap between low level data from maritime sensors and maritime domain semantics [25], for example to enhance the integration of maritime information (6), to model ships' behaviour [41], for patterns identification [2], abnormal behaviour detection [42], and prediction [7]. Indeed, semantics' representation and exploitation is preferably addressed at the application level, because existing semantic approaches and technologies are not adequate to address the requirements of multi-mission and multi-task MSAs. In particular, approaches for semantic multi-domain interoperability able to integrate heterogeneous information sources (e.g., surveillance data, weather and ocean data, registers, bulletins) and combine multi-representation formalisation of multiple contexts need to be developed.

Regarding the integrated exploitation of disparate data sources in the maritime domain, semantic representation of maritime information in multi-scale and at multiple granularity levels brings new bussiness opportunities as well as new research challenges. For instance, processing maritime data as linked stream data requires integration and joint processing of this data with quasi-static data from the Linked Data Cloud or other open data sources, in soft-real-time, that usually are at different scales and granularity levels. As additional examples, data from Earth Observation sensors and VMS data are at a lower temporal resolution than surveillance data from VTS radar and AIS, which have revisit times of few minutes; VTS radar spatial resolution is poor compared to GPS position accuracy from AIS, which is assumed to be around 10m; freely available meteorologic data have spatial resolution of few kilometres, and estimated and measured environmental variables are provided with hourly and daily means. In addition, using open data sources in the maritime domain is a challenge itself because of the different policies of European countries regarding the provision of environmental and other marine data to the users, that, depending on the data sources, can be classified at national level and not freely distributable at the necessary resolution.

2.6 Discussion

Support to real-time (semantic) integration, storage and spatio-temporal querying of disparate maritime data sources enabling scalability for massive amounts of dynamic data is a challenging task. A series of specific challenges are as follows:

- producing a scalable, fault-tolerant framework for cross-streaming data integration and processing of maritime data from multiple streaming sources, via the real-time computation of data synopses, achieving high rates of data compression;
- reconstruction of vessel trajectories and computation of events and multi-scale visualizations of data and patterns via advanced analytics techniques;
- incremental integration of maritime data, allowing advanced management and query answering of spatio-temporal data;
- automatic, real-time semantic annotation and linking of maritime data towards generating coherent views on integrated cross-streaming and archival data;
- efficient distributed management and querying of integrated trajectory and contextual data.

3. EVENT PATTERN DETECTION AND TRAJECTORY ANALYSIS

3.1 Event detection

Detection of anomalous vessel movements and analysis of suspicious vessel trajectories are crucial assets for improving the security of vessel traffic. The range of possible events of interest is very large, from detecting vessels in distress and collisions at sea to discovering illegal fishing and any other illicit activities occurring at sea such as contrabands and smuggling.

Detecting events and patterns of interest in the maritime domain requires, as a first step, correlating vessel trajectories with data expressing entities' characteristics, geographical information, weather data, patterns of mobility in specific areas, regulations, intentional data (e.g. planned routes) etc., in a timely manner, while addressing the challenges described in the previous section. As part of the data integration task, the annotation of trajectories with contextual data is expected to provide a more robust solution towards the semantic annotation of trajectories [34], the interlinking of events with trajectories and with other contextual data that are of particular importance for maritime domain awareness and decision-making. These present major challenges as they do concern both archival and streaming data, and they do require dealing with spatio-temporal features at multiple scales and dimensions (e.g., for determining the similarity among trajectories, or for relating events to trajectory segments) [4] .

The specific challenges of event detection and suspicious pattern identification in the maritime domain are driven by MSA objectives. In particular, the development of early warning anomaly detection algorithms supporting maritime operators in the identification of the potentially suspicious of dangerous activities in the maritime areas under surveillance encompasses many challenges, such as:

- real-time reconstruction of vessel trajectories, supported by real-time analysis of multiple and voluminous streams of data on possibly conflicting vessel positions;
- algorithms for the prediction of anticipated vessel trajectories at different time scale, which is fundamental to achieve early warning maritime monitoring;
- machine learning methods supporting the identification and the formalization of events and patterns that are of interest to maritime security operators, able to observe and learn from their behaviour;
- algorithms for complex event (and outlier) recognition and prediction in real-time, dealing with heterogeneous, fluctuating and noisy voluminous data streams of moving entities in large geographic areas, taking advantage of data analytics results over archival data.

3.2 Visual analytics

MSA may greatly benefit from the development of Visual Analytics (VA) methods oriented to the maritime domain. VA methods, being more oriented than traditional analysis approaches towards addressing human factors and enhancing user perception, may help obtaining better analysis results through a more effective integration of unformalized operative knowledge and expertise, which are of fundamental importance in surveillance activities [30]. Specific VA research challenges are as follows:

- interactive data exploration of both archival (data-at-rest) and streaming (data-in-motion) spatio-temporal data, with varying levels of resolution and quality;
- exploration of real-time maritime moving entities integrating contextual and historical information at varying levels of resolution, supporting operators in early alerting validation;
- scalable spatio-temporal analytical querying, such as drill-down / zoom-in and on user-defined spatio-temporal regions of interest for surveillance;
- interactive pattern extraction (and assessment of data quality) considering both data-in-motion and data-at-rest, able to visually integrate information on sensor performance to validate early alerts obtained by the analysis tools;
- user-guided model building and validation, aiming at visual steering of modelling tools enabling interactive selection of model types, tuning model parameters, and analysis of model residuals in multiple dimensions, including space and time;
- building situation overview and situation monitoring, capable of computing an overall operational picture of mobility at desired scales and levels of detail, both in spatial and temporal dimensions. Monitoring needs to provide alarms and explanations if observations significantly deviate from models.

4. TOWARDS A MARITIME DECISION SUPPORT AND FORECASTING SYSTEM

The variety of data sources is expected to provide an improved MSA to the operator taking advantage of the complementarity and redundancy provided. For instance, the knowledge captured in databases of records of events like piracy events or incidents at sea, the lists of vessels of interest such as blacklisted vessels, may provide the relevant context to understand and explain some events of interests. If the processing and correlation of data and information from different sources and databases can also overcome for some incompleteness in databases and compensate for sparseness, it however may reveal some inconsistencies that need to be managed. For instance, ship information from the MarineTraffic[1] database may conflict with that from Lloyds'[2] : the length may differ slightly, or the flag may be different due to a lack of update in one source. In this regards, additional knowledge on sources' quality may help solving the issue. An example of estimating and exploiting the AIS reliability is proposed in [8], while for enhancing the reliability of AIS, the vessel identity verification method has been used by the US Coast Guard (USCG)'s Maritime Information for Safety and Law Enforcement (MISLE) and Vessel Documentation System (VDS) [44].

The correlation with social media information [3] can furthermore help in establishing links with external events for a better global picture. However, most of these sources contain natural language information (possibly in different languages), which needs to be automatically processed, analysed, interpreted and finally correlated with other data from physical sensors. The fusion of human generated information ("soft") with sensor data ("hard"), thus named "hard and soft fusion", has been recently widely addressed [28] and brings promising avenue to the MSA problem [17], in keeping the human at the core of the processing. Hence, the design of an efficient information system for Maritime Domain Awareness and decision support should consider the maritime data quality issues in their entirety and diversity to ideally resolve them or at least to not occult them and rather inform the operator of some possible output uncertainty.

Probabilistic databases are certainly a promising avenue for the maritime domain [3] [23], which allows to deal for instance with empty fields very common in marine data, approximate values or uncertain fields. Besides, the consideration of the open-world assumption is unavoidable if one wants to provide a realistic outcome to the user [9]. Indeed, the AIS database clearly violates the closed-world assumption since, according to Windward [43], 27% of ships do not transmit data at least 10% of the time ('go dark'). Consequently, querying for instance *rendez-vous* events from an AIS database will return only those events reflected by the AIS data. Considering that anything which is not in the AIS database remains possible is thus crucial to maritime anomaly detection. Moreover, the extension to other uncertainty representations such as evidence or possibility theories is certainly desirable for maritime anomaly detection and event forecasting in order to cope with the different nature of uncertainty (probabilistic, subjective, vague, ambiguous, etc) due to the variety and poor veracity of the sources.

Although no clear guidelines exist so far for the selection of the appropriate uncertainty framework and aggregation (or fusion) rule, it is acknowledged that the choice depends on the nature, interpretation or type of uncertainty and information, and on the sources quality and independence [45]. Considering second-order

[1] http://www.marinetraffic.com

[2] http://www.lloydslistintelligence.com

uncertainty seems also unavoidable if one wants to properly account for the imperfection of data in the estimation of patterns-of-life, in developing approximate models of vessels' motion or in recognition of vessels, but also if one wants to communicate to the user faithful information. Interestingly, similar uncertainty challenges recently arose within the visualization community. A great challenge is to enable reasoning under uncertainty (in all its forms) uncertainty throughout the processes of sensemaking, decision-making, and action-taking [23].

In the aim of developing trustful and useful decision support systems, the human must be considered in his/her two main roles of (i) source of information and (ii) decision maker. The underlying challenges are then, on the one hand, to properly capture the human generated information including the associated uncertainty assessment so it can be meaningfully aggregated with other information from physical sensors or databases, and on the other hand, to ensure that the system outputs meaningful, interpretable and unambiguous results on which the user can take an informed decision. The design of information systems should provide a flexible architecture to ensure both the utility of the output provided and adaptability to dynamic and unforeseen events, and to changing user's needs. For instance, an explicit consideration of context provides an understanding of normalcy as a reference for anomaly detection (i.e., *pattern-of-life*) [40]. It helps detecting, and distinguishing between, spoofed information and deception, reducing the set of possible hypotheses (e.g., classes) for threat classification; it also provides information about sources' quality such as reliability or truthfulness.

Finally, the development of decision support systems for maritime event detection and forecasting should provide (1) the necessary simplicity to the processes by a judicious filtering of information suited to the users' needs, (2) the suitable flexibility and adaptability for the algorithms implementation by separating between the events of interest and their surrounding context, (3) the adequate uncertainty representation and processing considering the sources' quality and uncertainty's origin and (4) the expected human-system synergy for a better understanding of the system' outputs with associated explanations and simpler queries tuned to specific needs.

5. CONCLUSION

While most of current research in spatial databases and geographical information systems addresses issues often related to phenomena and practices related to the land domain, we believe that the maritime environment also provides many application opportunities and research challenges that still deserve to be addressed. This paper surveys a series of current computational issues still opened for a successful integration, manipulation and analysis of maritime information, with a specific focus on trajectories of moving objects at sea. We explore and suggest several research development directions that might contribute to a better use of voluminous and disparate sets of maritime data available so far. Due to the diversity and complexity of the problem, a successful solution should involve an integration of complementary contributions from different scientific domains, let us mention amongst many research areas ontology and conceptual data models at the data integration level, data mining and visual analytics for the ability to discover patterns within large volume of data, machine learning for streaming data, information fusion for the ability to combine information from different sources and deal with uncertainty, human factor and decision-aided systems. This is why we believe that not only many opportunities are still open for the extended database community, but also avenues for further experimentations and interactions with the maritime world at large.

6. ACKNOWLEDGMENTS

This work was supported by project datACRON, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 687591.

7. REFERENCES

[1] G. Andrienko, N. Andrienko and G. Fuchs, Understanding movement data quality. *Journal of Location Based Services*, 10(1), 31-46, 2016.

[2] H. Arenas, B. Harbelot and C. Cruz, A semantic analysis of moving objects using as a case study maritime voyages from eighteenth and nineteenth centuries. In *Proc. of GEOProcessing*, pp. 45-50, 2014.

[3] M.L. Ba, S. Montenez, R. Tang, and T. Abdessalem, Integration of web sources under uncertainty and dependencies using probabilistic XML. In *Proc. of Database Systems for Advanced Applications*, pp. 360-375, 2014.

[4] M. Balduzzi, A. Pasta and K. Wilhoit, A security evaluation of AIS automated identification system. In *Proc. of ACSAC.* pp. 436-445, 2014.

[5] S. Bothe, V. Manikaki, A. Deligianakis and M. Mock, Towards flexible event processing in distributed data streams. In *Proc. of EPForDM EDBT/ICDT Workshop*, 2015.

[6] S. Brüggemann, K. Bereta, G. Xiao and M. Koubarakis, Ontology-based data access for maritime security. In *Proc. of ESWC*, pp. 741-757, 2016.

[7] R. N. Carvalho , R. Haberlin , P. Cesar , G. Costa , K. B. Laskey and K. Chang, Modeling a probabilistic ontology for maritime domain awareness. In *Proc. of FUSION*, pp. 1-8, 2011.

[8] D. Ceolin, W. R. van Hage, G. Schreiber and W. Fokkink, Assessing trust for determining the reliability of information. In *Situation Awareness with Systems of Systems*, Springer, New York, 209-228, 2013.

[9] I.I. Ceylan, A. Darwiche and G. Van den Broeck, Open-world Probabilistic Databases. In *Proc. of KR*, pp. 339-348, 2016.

[10] F. Chirigati, H. Doraiswamy, T. Damoulas and J. Freire, Data polygamy: the many-many relationships among urban spatio-Temporal data sets. In *Proc. of SIGMOD*, pp. 1011-1025, 2016.

[11] G. Cugola and A. Margara, Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, 44(3), article 15, 2012.

[12] datACRON EU H2020 project: Big Data Analytics for Time Critical Mobility Forecasting'. URL: http://datacron-project.eu.

[13] D. Dubois, W. Liu, J. Ma and H. Prade, The basic principles of uncertain information fusion: An organised review of merging rules in different representation frameworks, *Information Fusion*, 32A, 12-39, 2016.

[14] O. Dunemann, I. Geist, R. Jesse, K.-U. Sattler, and A. Stephanik, A database-supported workbench for information fusion: INFUSE. In *Proc. of EDBT*, pp. 756-758, 2002.

[15] A. Eldawy and M. F. Mokbel, The era of big spatial data. In *Proc. of ICDE Workshops*, pp. 42-49, 2015.

[16] EMSA, Automated behaviour monitoring (ABM) algorithms – operational use at EMSA. In *Proc. of MKDAD Workshop*, pp. 12-16, 2016.

[17] R. Falcon, R. Abielmona, S. Billings, A. Plachkov, and H. Abbass, Risk management with hard-soft data fusion in maritime domain awareness. In *Proc. of CISDA*, pp. 1-8, 2014.

[18] M. Fidali and W. Jamrozik, Concept of database architecture dedicated to data fusion based condition monitoring systems. In *Proc. of BDAS*, CCIS 424, pp. 515–526, 2014.

[19] M. Guerriero, P. Willett, S. Coraluppi and C. Carthel, Radar/AIS data fusion and SAR tasking for Maritime Surveillance. In *Proc. of FUSION*, 2008.

[20] S. Gurajada, S. Seufert, I. Miliaraki and M. Theobald: TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *Proc. of SIGMOD*, pp. 289-300, 2014.

[21] S J. N. Hughes, M. D. Zimmerman, C. N. Eichelberger and A. D. Fox, A survey of techniques and open-source tools for processing streams of spatio-temporal events. *In Proc. of IWGS@SIGSPATIAL*, 6:1-6:4, 2016.

[22] M. F. Husain, J. P. McGlothlin, M. M. Masud, L. R. Khan and B. M. Thuraisingham, Heuristics-based query processing for large RDF graphs using cloud computing. *IEEE Trans. Knowl. Data Eng*. 23(9), 1312-1327, 2011.

[23] A.-L. Jousselme and P. Maupin, Uncertainty representations for information retrieval with missing data. In *Fusion Methodologies in Crisis Management*, Springer, 87-104, 2016.

[24] K. Kyzirakos, M. Karpathiotakis, and M. Koubarakis. Strabon: A semantic geospatial DBMS. In *Proc. of ISWC*, pp. 295-311, 2012.

[25] A. Leadbetter, T. Hamre, R. Lowry, Y. Lassoued and D. Dunne, Ontologies and ontology extension for marine environmental information systems. In *Proc. of EISSAP* at *EnviroInfo*, Bonn, 2010.

[26] D. Le-Phuoc, H.N. Mau Quoc, H.N. Quoc, T.T. Nhat and M. Hauswirth, The graph of things: a step towards the Live Knowledge Graph of connected things. *Journal of Web Semantics*, 37, 25-35, 2016.

[27] J. Liagouris, N. Mamoulis, P. Bouros, and M. Terrovitis. An efective encoding scheme for spatial RDF Data. *PVLDB*, 7(12), 1271-1282, 2014.

[28] J. Llinas, Challenges in information fusion technology capabilities for modern intelligence and security problems. In *Proc. of EISIC*, pp. 89-95, 2013.

[29] J. Lu and R. H. Guting, Parallel secondo: practical and efficient mobility data processing in the cloud. In *Proc. of Big Data*, IEEE Press, pp. 17-25. 2013.

[30] A. M. MacEachren, Visual analytics and uncertainty: It's not about the data. In *Proc. of EuroVis Workshop on Visual Analytics*, 2015.

[31] Maritime CISE, A Common Information Sharing Environment for Maritime Surveillance in Europe https://webgate.ec.europa.eu/maritimeforum/en/frontpage/1046 (accessed in June 2016)

[32] A.-C. Ngonga Ngomo. On link discovery using a hybrid approach, *J.Data Semant*, 1, 203-217, 2012.

[33] N. Papailiou, I. Konstantinou, D. Tsoumakos and N. Koziris: H2RDF: adaptive query processing on RDF data in the cloud. WWW (Companion Volume), 397-400, 2012.

[34] C. Parent, S. Spaccapietra, C. Renso, G. Andrienko, N. Andrienko, V. Bogorny, M. L. Damiani, A. Gkoulalas-Divanis, J. Macedo, N. Pelekis, Y. Theodoridis and Z. Yan, Semantic Trajectories Modeling and Analysis. *ACM Computing Surveys*, 45(4), article 42, 2013.

[35] K. Patroumpas, A. Artikis, N. Katzouris, M. Vodas, Y. Theodoridis and Y. Pelekis, Event recognition for maritime surveillance. In *Proc. of EDBT*, pp. 629-640, 2015.

[36] C. Ray, C. Iphar, A. Napoli, R. Gallen and A. Bouju, DeAIS project: Detection of AIS spoofing and Resulting Risks. In *Proc. of OCEANS'15 MITS/IEEE*, 2015.

[37] L. Salmon and C. Ray, Design principles of a stream-based framework for mobility analysis. *GeoInformatica*, 20(2), 1-25, 2016.

[38] S. Shekhar, V. Gunturi, M. R. Evans, and K. Yang, Spatial big-data challenges intersecting mobility and cloud computing. In *Proc. of MobiDE*, pp. 1-6, 2012.

[39] Silk (http://wifo5-03.informatik.uni-mannheim.de/bizer/silk) by Univ. of Mannheim

[40] L. Snidaro, J. Garcia and J. Linas, Context-based information fusion: a survey and discussion. *Information Fusion*, 25, 16-31, 2013.

[41] W. R. van Hage, V. Malaisé, G. de Vries, G. Schreiber, and M. van Someren, Combining ship trajectories and semantics with the simple event model (SEM). In *Proc. of EiMM*, pp. 73-80, 2009.

[42] A. Vandecasteele and A. Napoli. An enhanced spatial reasoning ontology for maritime anomaly detection. In *Proc. of IEEE SOSE*, pp. 247-252, 2012

[43] Windward, AIS Data on the High Seas: An Analysis of the Magnitude and Implications of Growing Data Manipulation at Sea, http://www.windward.eu/wp-content/uploads/2015/02/AIS-Data-on-the-High-Seas-Executive-Summary-Windward-October-20-2014.pdf (accessed in June 2016).

[44] D. Winkler, Enhancing the reliability of AIS through vessel identity verification, US DOT Datapalooza, https://www.fhwa.dot.gov/2015datapalooza/presentations/Safety.4_Winkler.pdf (accessed in June 2016).

[45] Z. Yu, Y. Liu, X. Yu, and K. Q. Pu, Scalable distributed processing of K-nearest neighbor queries over moving objects. *IEEE Trans. Knowl. Data Eng.*, 27(5), 1383-1396, 2015.

[46] X. Zhang, L. Chen, Y. Tong and M. Wang: EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In *Proc. of ICDE*, pp. 565-576, 2013.

[47] K. Zeng, J. Yang, H. Wang, B. Shao and Z. Wang: A Distributed Graph Engine for Web Scale RDF Data. *PVLDB* 6(4), 265-276, 2013.

DeepSea: Progressive Workload-Aware Partitioning of Materialized Views in Scalable Data Analytics

Jiang Du
University of Toronto
jdu@cs.toronto.edu

Boris Glavic
Illinois Institute of Technology
bglavic@iit.edu

Wei Tan
IBM T. J. Watson Research Center
wtan@us.ibm.com

Renée J. Miller
University of Toronto
miller@cs.toronto.edu

ABSTRACT

Selective materialization of intermediate query results as views is an effective method for improving query performance. In this paper, we extend this technique to adaptively partition views based on the access patterns of a workload. That is, we collect information about the selection conditions of queries at runtime and utilize this information to determine fragment boundaries for the initial partitioning when materializing a view. Furthermore, we refine view partitions over time based on the selection conditions of incoming queries. We present a novel cost-benefit model for partitioned views, as well as a candidate view and fragment selection approach - both of which exploit the nature of partitioned views by taking the correlation among view fragments into account. Furthermore, we present DeepSea, an implementation of these techniques built on top of Hive. Our experimental evaluation demonstrates the effectiveness of partitioned views, improving performance by up to an order of magnitude compared to state-of-the-art approaches.

1. INTRODUCTION

The use of materialized views is a common technique to improve the performance of query workloads [21]. The questions of what to materialize, when to materialize, and when to use a view have been well studied. The same is true for other automated physical design techniques such as index and partition selection. Proper physical design for base tables, e.g., horizontal partitioning, often significantly improves the performance of queries [23]. In modern SQL systems built on-top of distributed dataflow engines (e.g., Hadoop [1]), issues of physical design, including partitioning of large files, are paramount to the performance of the system. Furthermore, intermediate results are often materialized for fault tolerance purposes and these results can be utilized as materialized views to answer future queries [12]. While each of these techniques has been studied intensively, we are the first to study the combination of materialized view selection and horizontal partitioning.

The major advantage of creating a partitioned view from an intermediate query result is that future queries with selection conditions over the partition attribute can be answered efficiently by accessing a subset of the view's fragments. However, partitioning a view increases the cost of view creation. Furthermore, new challenges arise because we have to decide when to partition a view, how to select fragment boundaries (within a partitioning), when to repartition, and what fragments to evict to save space. We address these challenges in this work.

Online View Selection. A view selection algorithm that is based on a query workload is called *adaptive*. Adaptive (or workload-aware) materialization and partitioning of views may be done at design-time or at runtime. That is, either a complete workload is given and the *view selection* algorithm determines which views to materialize and how to partition them offline, or the algorithm works in an *online* fashion making decisions based on the history of queries that have been processed so far. While online materialized view selection has been studied [24], we are the first to consider the online adaptation of partitioning choices for views. Our partitioning strategy is motivated by two important characteristics of real-life data analytic workloads: 1) data access is often not distributed uniformly over the domain of a selection attribute and 2) access patterns evolve as the interests of users change over time.

Non-Uniform Distribution of Access. Figure 1 shows the access distribution for a real analytic workload over the Sloan Digital Sky Survey dataset (SDSS) [2]. The figure shows the selection ranges on attribute *ra* of table *PhotoPrimary* for queries submitted to SDSS between March 8, 2010 and March 8, 2011. Note that there are ranges that are rarely queried and others that are very frequently queried. Clearly, *adaptive partitioning* can improve query performance. We use the range conditions of queries to adjust fragment (partition) boundaries with the effect that *hot spots* are covered by relatively small fragments and less frequently accessed data are covered by fewer and larger fragments. This has the advantage of focusing the effort of partitioning on the parts of the data which will give us the most benefit. Queries accessing hot spots can be answered using small fragments without touching unwanted ranges of the view. Furthermore, using this approach we avoid paying the cost of partitioning data that is accessed infrequently.

Evolving Access Patterns. In addition to being non-uniform, real workloads are not static, but rather access patterns may shift over time. Figure 2 shows how the selection ranges of SDSS queries over attribute *ra* of table *PhotoPrimary* evolve over the sequence of the first 10,000 queries containing such a selection, starting from March 8, 2010. The vertical line near query 1,000 means that one or more queries have selected the whole domain of attribute *ra*. The figure shows that the first 3,000 queries focus mainly on the range between 200 and 300 degrees. Later in the workload, a large number of queries focus on values around 100 degrees.

To accommodate evolving access patterns, we make decisions on how to partition a view online as queries arrive. We create a mate-

Series ISSN: 2367-2005

10.5441/002/edbt.2017.19

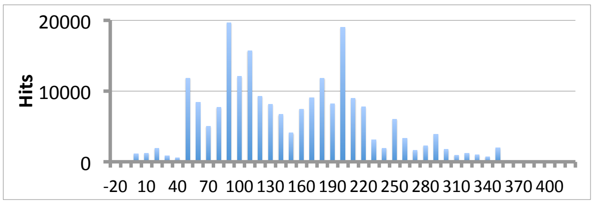


Figure 1: Histogram of selection ranges on SDSS

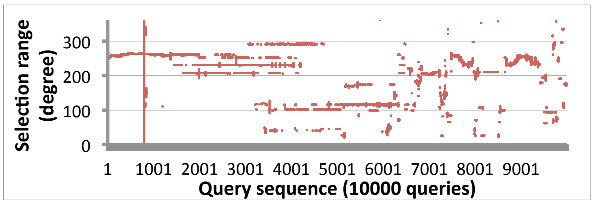


Figure 2: Evolution of selection ranges on SDSS

rialized view with an initial partitioning once we have determined that there is enough evidence that the creation of the view will benefit the current workload. We *progressively* refine fragment boundaries based on the selection conditions of incoming queries. Our progressive partitioning, coupled with a cost-based view and fragment eviction policy, allows us to adapt to evolving workloads. A view's competitiveness according to this cost model is based on its observed *benefits* (an estimate of the runtime that would have been saved if the view were to be materialized), its *creation cost* (the runtime overhead of materializing and partitioning the view), and its *storage size*. Importantly, we apply a decay function to timeout view benefits over time. This ensures that after a shift in the workload, views that are no longer useful for the current access pattern will eventually be replaced with views that fit the new pattern.

Partitioned Materialized View Pool Size. Typically, the storage space allocated for materialized views is not unlimited. We analyzed a BigBench workload [13] and found that if we materialize all intermediate join results as views, the total storage required is four times the size of the BigBench base tables. Of course, for evolving workloads, the number of materialized views and fragments would continue to increase and not all views will continue to provide a benefit to queries. Jain et al. [20] show the importance of a good *view selection strategy* for real-life applications: the savings that can be achieved with a small materialized view pool are similar to the savings that can be achieved with a large pool size as long as a good view selection strategy is applied. An important benefit of partitioned views is the finer granularity of control on view and partition selection: we can individually evict the fragments of a partitioned view that are unlikely to be used in the future.

Correlated Fragments. Given a finite amount of space for storing views, we present a novel strategy for selecting what fragments of a view to keep. Typically, decisions on whether to keep or evict a view are made independently for each view [15]. However, the benefits that different fragments of a partitioned view provide to a workload are not independent of each other. Returning to Figure 1, observe that ranges which are accessed often (ranges with many *hits*) tend to have neighbors with many hits (which are also accessed often). We find similar patterns for other attributes of different SDSS tables: parts of the domain of an attribute that are close to hot spots have a higher chance of being hit in the future than parts that are further away from hot spots. We present a new probabilistic model based on this correlation to determine when a fragment of view should be evicted. Our model treats a hit to a fragment as a sample from a probability distribution. We determine the normal distribution that has the maximum likelihood to have produced the sample and use this distribution for fragment selection.

Overlapping Fragments. Figure 2 also hints at a common pattern for selection ranges. A partition containing a few large fragments (for cold spots) and several small fragments (located at hot spots) may work well for some time, but as the workload evolves, there is a need to split a large fragment as additional queries begin to access it. This split incurs high write cost for repartitioning because if a fragment is split, its whole content needs to be read and written to disk. We present a solution that permits overlapping fragments. Rather than reading and writing the large fragment, we create a small fragment that overlaps the large fragment.

Contributions. Our main contributions are as follows.

- *Progressive, adaptive partitioning of materialized views.* We propose the first algorithm for progressively partitioning materialized views that adapts online to changes in a query workload.
- *Exploitation of fragment correlations*. Based on our study of real-life workloads, we present a novel cost-benefit model for view fragments and candidate selection that takes the correlation among fragments of a partition into account.
- *Overlapping fragments*. We allow overlapping fragments and show that they can reduce the cost of view creation especially over evolving workloads.
- *DeepSea*. We present DeepSea, an implementation of our techniques in Hive [27].
- *Evaluation*. We demonstrate DeepSea's effectiveness using a query workload modelled after a real workload from SDSS [2] and workloads from BigBench [13].

The remainder of the paper is organized as follows. We discuss related work in Section 2, introduce preliminaries in Section 3, and formally state the problem addressed in this work in Section 4. We give an overview of our approach in Section 5. We then present how to select view candidates in Section 6, how to select what to materialize and how to partition in Section 7, and how to answer queries using partitioned materialized views in Section 8. Afterward, we discuss the implementation of DeepSea in Section 9 and present our experimental evaluation in Section 10.

2. RELATED WORK

There are several lines of work related to our approach: answering queries using views; reusing intermediate query results; (online) self-tuning techniques for physical database design; database cracking; and semantic caching.

Answering Queries Using Views. Answering queries using materialized views has been studied intensively [3, 21]. Given a set of views and a query, computing the least expensive plan for the query using the views is computationally hard, because query containment checks are required to determine whether a query can be computed from a view. Query containment for bag semantics (SQL) is undecidable, even for restricted query classes (union of conjunctive queries). As a consequence, practical approaches for *logical matching* (i.e., determining whether a view can be used to answer a query independent of the query syntax) usually apply sufficient conditions for matching that are decidable or even in PTIME [14, 29]. Goldstein and Larson [14] present a lightweight algorithm that is integrated with a transformation-based optimizer and uses a cost model to determine the best rewriting. We have extended this approach to support matching fragments of partitioned views.

Reusing Intermediate Results. Although materialization has been studied extensively for relational databases [3, 16, 21], distributed systems such as Hadoop have different characteristics that need to be explored and exploited. ReStore [12] materializes intermediate results of MapReduce jobs for reuse in future queries. Perez and Jermaine [24] exploit salient features of SQL-on-Hadoop systems including immutable data, abundant storage to accommodate materialized views, and excessive materialization of intermediate results that enables generating materialized views as a by-product of answering queries. The approach optimizes queries to produce intermediate results that, if materialized as views, would improve performance for past queries. If past queries are indicative of future queries then this would result in a speed up for future queries. We also gather knowledge about a workload to guide materialization, but in addition investigate partitioning for materialized views. The Nectar system [15] caches and reuses results of DryadLINQ/Dryad computations. ReStore and Nectar only perform physical matching, i.e., a view matches a sub-query if they are computed using the same expression. As explained previously, we decide to use logical matching which greatly improves the potential for reuse. Reuse of intermediate query results has also been studied for main memory DBMS such as MonetDB [19, 22]. This approach uses physical matching of operators except for selections where subsumption of range restrictions is considered, e.g., the result of a selection on $A < 5$ is a superset of the result of a selection on $A < 3$, and thus a query with selection $A < 3$ can be rewritten by using a materialized view whose selection is $A < 5$. Similar to ReStore and in contrast to automated materialized view and index selection approaches for relational databases, our approach significantly reduces view creation cost by considering intermediate query results as candidates for materialized view creation. In addition, whenever possible we use intermediate results that are materialized anyways by the MapReduce engine (e.g., at the end of a reduce phase).

Automated Physical Design. Automated tuning [10] is a rich field including: partitioning [23, 25], index selection [6, 26], and materialized view selection [4, 5, 8]. Adaptive index selection creates and drops indexes on-the-fly [6, 26]. Given a constraint on storage space, the idea is to monitor incoming queries and profile the performance gain for each index and then create the most promising ones. Adaptive materialized view selection [4, 5, 8] shares the same philosophy. Both index and materialized view selection use the DBMS optimizer's cost model to evaluate the benefits of an index or view without actually creating it. Bruno and Chaudhuri [9] have explored online index selection that is 3-competitive. However, this bound only holds for single index candidates. In contrast to these approaches we do not assume a sophisticated optimizer. Our solution also repartitions data on-the-fly, as a by-product of query answering. The H_2O system [7] supports multiple storage layouts, i.e., *columnar*, *row* and *group of columns*. At run-time, the system decides which layout to use for which part of the data, and continuously evolves the storage layout and data access strategy. In constrast to H_2O, we focus on horizontal partitioning of data in a distributed environment, and address the size requirement of materialized view pool.

Database cracking. Database cracking [18], i.e., adaptive and progressive indexing, incrementally builds an index structure over a table based on access patterns of queries. There is a rich body of work on enhancements of cracking such as the study of robustness and adaptiveness to dynamic workloads [17]. A similarity between cracking and our approach is that they both incrementally refine physical designs based on selection conditions in queries. In contrast to cracking, DeepSea focuses on horizontal partitioning of materialized views and makes cost-based decisions on whether to refine a partition.

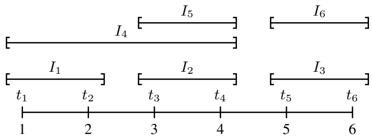
Semantic caching. Semantic caching [11] studies how to reuse subsets of input tables that are stored in a client-side cache. Each entry in the cache is described as a logical constraint (selection condition) providing a semantic description of the content of a cache entry. When a query is submitted to the client and can be answered (partially) using the cache, only a "remainder query" will be sent to the server to fetch the results that do not exist in the client's cache. Similar to DeepSea, intermediate results are reused and are reorganized based on access patterns. However, semantic caching only considers caching of the results of selections over base tables (we consider caching of a view that is partitioned on a selection attribute) and does not allow cached regions to overlap.

3. PRELIMINARIES

We now review the concept of horizontal partitioning. We use R, $S, \ldots$ to denote relations, $A, B, \ldots$ to denote attributes, and $\mathcal{D}(A)$ to denote the domain of attribute A. We call an attribute A *ordered* if there exists a total order $\leq_A$ over $\mathcal{D}(A)$. Only ordered attributes are considered as keys for horizontal partitioning.

Horizontal Partitioning. Horizontal partitioning splits the tuples of a relation into a set of disjoint fragments - each fragment holds the data for a range of values of the partition key (the attribute on which we partition). The union of these fragments equals the original relation.

DEFINITION 1 (HORIZONTAL PARTITIONING). *Let R be a relation and A an ordered attribute from R's schema. Consider a set $\mathcal{I} = \{I_1, \ldots, I_n\}$ of intervals where $I_i \subseteq \mathcal{D}(A)$. The* fragmentation *$\mathbb{P}_{\mathcal{I}}(R.A)$ of R on A according to $\mathcal{I}$ is the set of fragments $F_i \subseteq R$ defined as $F_i = \{t \mid t \in R \wedge t.A \in I_i\}$. If $\bigcup_{I \in \mathcal{I}} I = \mathcal{D}(A)$ and $\forall i, j : I_i \cap I_j = \emptyset$ then $\mathbb{P}_{\mathcal{I}}(R.A)$ is called a horizontal partition.*



EXAMPLE 1. *Assume a relation R has 6 tuples $\{t_1, t_2, t_3, t_4, t_5, t_6\}$ where the value of attribute A for tuple t_i is i. The domain $\mathcal{D}(A)$ of A is $\{1, \ldots, 6\}$. Consider a set $\mathcal{I}$ of three intervals $I_1 = [1, 2]$, $I_2 = [3, 4]$, and $I_3 = [5, 6]$ as shown above. A partitioning based on these intervals would result in fragments $F_1 = \{t_1, t_2\}$, $F_2 = \{t_3, t_4\}$, and $F_3 = \{t_5, t_6\}$. The fragmentation $\mathbb{P}_{\mathcal{I}}(R.A)$ is a horizontal partition of R according to A. Consider a second set of intervals $\mathcal{I}'$ containing $I_4 = [1, 4]$, $I_5 = [3, 4]$, and $I_6 = [5, 6]$. The fragmentation according to $\mathcal{I}'$ results in fragments $F_4 = \{t_1, t_2, t_3, t_4\}$, $F_5 = \{t_3, t_4\}$, and $F_6 = \{t_5, t_6\}$. This fragmentation $\mathbb{P}_{\mathcal{I}'}(R.A)$ is not a horizontal partition of R, because of the overlap between I_4 and I_5. Finally, $\mathcal{I}'' = \{I_4, I_6\}$ is again a horizontal partition of R.*

Overlapping Partitioning. It is sometimes beneficial to relax the disjointness requirement by allowing fragments to overlap. We call such a fragmentation an *overlapping partitioning*.

DEFINITION 2 (OVERLAPPING PARTITIONING). *Let R be a relation and A one of its attributes. We call a fragmentation $\mathbb{P}_{\mathcal{I}}(R.A)$ an overlapping partitioning iff $\bigcup_{I \in \mathcal{I}} I = \mathcal{D}(A)$.*

EXAMPLE 2. *Figure 3 illustrates why it may be beneficial to allow fragments to overlap. Assume that a query* Q1 *acccesses a range $[a, b]$ and that based on this access pattern we have decided*

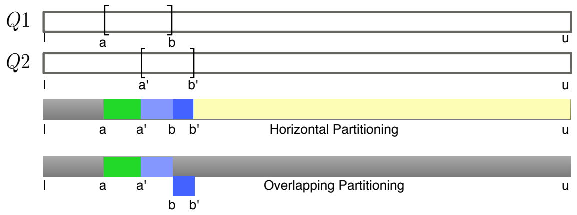


Figure 3: Overlapping fragments in progressive partitioning

to create a partition with three fragments $[l, a)$, $[a, b]$, *and* $(b, u]$. *A subsequent query* Q2 *accesses data in the range* $[a', b']$. *Note that* b *and* b' *are close to each other. Adaptive horizontal partitioning may create four new fragments based on* Q2 *by splitting the previously created fragments into* $[a, a')$, $[a', b]$, $(b, b']$ *and* $(b', u]$. *If we allow fragments to* overlap, *then we can avoid creating the fragment* $(b', u]$ *because no query has accessed data from this fragment yet. Instead, we create a fragment* $(b, b']$ *and keep the fragment* $(b, u]$ *that was created based on* Q1. *This avoids writing a large fragment that may not be accessed by future queries at the cost of additional storage for* $(b, b']$.

4. PROBLEM STATEMENT

We now state the problem addressed in this work: how to maintain a set of partitioned views (the *materialized view pool*) in an online fashion in order to maximize query performance.

Configuration. A configuration C models the current content of the materialized view pool. It consists of the set of views $\mathcal{V}$ that are currently in the pool and a mapping $\mathcal{P}$ that associates each view V and one of its attributes A with a set of intervals describing the current partitioning of the view on this particular attribute. Note that we permit multiple partitions of a view to be stored in the pool as long as these partitions are on different attributes. We define $\mathcal{P}(V, A) = \emptyset$ if view V has not been partitioned on attribute A yet.

DEFINITION 3. *A configuration* C *is a pair* $(\mathcal{V}, \mathcal{P})$ *where* $\mathcal{V}$ *is the set of views materialized in the pool and* $\mathcal{P}$ *is a mapping that associates with each view* $V \in \mathcal{V}$ *and an attribute* A *in the schema of* V *a set of intervals* $\mathcal{I}$ *over the domain* $\mathcal{D}(A)$ *of* A. *We use* $S(C)$ *to denote the total storage size of the views in configuration* C.

Problem Definition. In this work, we assume a query-only workload, i.e., no updates. We address the following problem: given a *workload* $\mathcal{Q} = Q_1, \ldots, Q_n$ of queries to be executed that is unveiled one query at a time and a *pool size limit* S_{max} (maximal storage to be used for views), choose a sequence of configurations $\mathcal{C} = C_1, \ldots, C_n$ in order to minimize the total execution time of the workload plus the time spent on view creation $\textsc{Cost}(\mathcal{Q}, \mathcal{C}) = \sum_{i=1}^{n} \textsc{Cost}(Q_i, C_i) + \sum_{i=1}^{n-1} \textsc{Cost}(C_i, C_{i+1})$. Here $\textsc{Cost}(Q, C)$ denotes the cost of executing query Q given the set of views C and $\textsc{Cost}(C_i, C_{i+1})$ denotes the cost of creating configuration C_{i+1} from configuration C_i. We require $C_1 = \emptyset$, i.e., no views have been created before the workload execution. We are interested in a restricted version of this problem where new views and refinements of partitions have to be based on the currently executed query Q_i, i.e., only views and fragments corresponding to intermediate results of this query ($\mathcal{V}_{cand}(Q_i)$ and $\mathcal{P}_{cand}$, defined in Section 6) are considered as candidates to be added to C_{i+1}. Given these preliminaries we can state the online partitioned view selection problem as follows.

DEFINITION 4 (ONLINE PARTITIONED VIEW SELECTION). *Given a pool size limit* S_{max} *and workload* $\mathcal{Q} = Q_1, \ldots, Q_n$ *that is unveiled one query at a time, incrementally determine the sequence of configurations* $\mathcal{C} = C_1, \ldots, C_n$ *that minimizes*

$$\textsc{Cost}(\mathcal{Q}, \mathcal{C}) = \sum_{i=1}^{n} \textsc{Cost}(Q_i, C_i) + \sum_{i=1}^{n-1} \textsc{Cost}(C_i, C_{i+1})$$

subject to

1. $C_1 = \emptyset$
2. $C_{i+1} - C_i \subseteq \mathcal{V}_{cand}(Q_i) \cup \mathcal{P}_{cand}$ *for all* $i \in \{1, \ldots, n-1\}$
3. $S(C_i) \leq S_{max}$ *for all* $i \in \{1, \ldots, n\}$

The online partitioned view selection problem is difficult for several reasons. First, this is an *online* problem: for each incoming query Q_i, we must decide which partitioned views or fragments to create and which to evict from the pool without knowing the remaining sequence of queries from the workload. There is abundant literature for online algorithms that provide worst-case guarantees. An online algorithm is said to be *k-competitive* if its result is at most of a factor k worse than the solution computed by an optimal offline algorithm (an algorithm which has access to the whole input). However, the competitiveness factor of such algorithms for search space sizes encountered in our problem are too high to be of any practical relevance. Even if we were to consider the offline version of the problem, we cannot hope for an optimal solution because of the undecidability of query answering with views.

Given these constraints we strive for a principled yet scalable solution that applies a carefully selected set of heuristics for each of the sub-problems of *determining view and partition candidates*, *view and partition selection* (determine the next configuration), and *view and partition matching* (determining whether a partitioned view can be used to answer a query). The main idea underlying our approach is that a solution should take hints provided by queries in the workload into account when deciding which intermediate query results to materialize and how to partition them.

5. SOLUTION OVERVIEW

Algorithm 1 gives a high-level view of the approach we use to process a query. The input to the algorithm is a query Q, view configuration C, and view statistics STAT. In the first step we determine which views and fragments (in the pool or not) can be used to answer the query (Section 8). The result of this step is a set $Rewr(Q)$ of possible rewritings of the input query which use the views. We then update the statistics kept for partitioned views to record that some views/fragments can be used to answer the query. Afterwards, among the rewritings that only use queries which are currently in the pool (C) we determine the rewriting Q_{best} with the lowest expected cost. Now that we have chosen a "plan" for the query (Section 6), we determine which of the intermediate results of the query are viable candidates to be stored as materialized views ($\mathcal{V}_{cand}$) and how to partition them ($\mathcal{P}_{cand}$). Note that even if a view $V \in \mathcal{V}_{cand}$ already exists and is partitioned, we may still produce fragment candidates for it (e.g., splitting an existing fragment to create a refined partition). Given such sets of candidates we add them to the set of partitioned views for which we want to keep statistics (using an initial rough estimate of their costs and benefits). The next step, described in more detail in Section 7, is to determine which of these candidates should be materialized during the execution of Q_{best} and, if necessary, which views to evict from the current configuration C to make space for these new views (recall that we limit the pool size by S_{max}). Once we have selected the views $\mathcal{V}_{sel}$ and fragments $\mathcal{P}_{sel}$ to create, we instrument the query Q_{best} to materialize intermediate results (and partition them if need

Algorithm 1 `ProcessQuery` $(Q, C, \textsc{Stat})$

```
Input  : Query $Q$, View Configuration $C$, View Statistics STAT
Output : Updated configuration $C$ and statistics STAT

$Rewr(Q)$ = computeRewritings($Q, C, \textsc{Stat}$)
updateStats($Rewr(Q), \textsc{Stat}$)
$Q_{best}$ = selectRewriting($Rewr(Q)$)
$(\mathcal{V}_{cand}, \mathcal{P}_{cand})$ = computeViewCand($Q_{best}, C, \textsc{Stat}$)
addCandidates($\mathcal{V}_{cand}, \mathcal{P}_{cand}, \textsc{Stat}$)
$(\mathcal{V}_{sel}, \mathcal{P}_{sel})$ = viewSelection($\mathcal{V}_{cand}, \mathcal{P}_{cand}, C$)
$Q_{best}^{instr}$ = instrumentQuery($Q_{best}, \mathcal{V}_{sel}, \mathcal{P}_{sel}$)
executeQuery($Q_{best}^{instr}$)
updateStats($\mathcal{V}_{cand}, \mathcal{P}_{cand}, \textsc{Stat}$)
```

be). We then execute the instrumented query Q_{best}^{instr} and return its result to the user. Finally, we update the statistics for all candidates based on the information gained by executing Q_{best}, e.g., we now have precise measurements for the size of candidate views.

6. VIEW AND PARTITION CANDIDATES

We now discuss how our approach determines which views and fragments to create for a given query Q and configuration C. Our creation process operates in two steps: first we determine for which views and fragments we have gathered enough evidence to materialize them and then based on this subset of candidates we determine the next configuration based on the "value" of a view or a fragment using the statistics that we keep.

View and Fragment Statistics. For each view or fragment candidate, no matter whether materialized in the pool or not, we store statistics such as its size S, the estimated cost of creating it (COST), the set of timestamps when this view could have been used to answer a query (T), and a list of potential savings associated with each such timestamp (B). B and T together with a decay function that times out benefit as mentioned in Section 1, are used to compute the *benefit* of a view. For fragments we only record T and S since the benefit can be inferred based on its size and the saving of the view this fragment belongs to. Similarly, COST of a fragment is determined based on COST for its view.

DEFINITION 5. *The view statistics* STAT *is a triple* $(\mathcal{V}_{\textsc{Stat}}, \mathcal{P}_{\textsc{Stat}}, \Sigma)$ *where* $\mathcal{V}_{\textsc{Stat}}$ *is a set of views,* $\mathcal{P}_{\textsc{Stat}}$ *is a mapping as in* C *that associates each view and attribute in its schema with a set of fragment intervals, and* Σ *maps each view in* $\mathcal{V}_{\textsc{Stat}}$ *and fragment in* $\mathcal{P}_{\textsc{Stat}}$ *to a tuple* $(S, \textsc{Cost}, T, B)$ *respective* (S, T).

6.1 View Candidates

We first notice that certain relational operators are less likely to provide results that can be reused or the reuse of such an operator's result would not result in significant performance improvement. We consider the intermediate results of the following operators as candidates: join, aggregation, and projection. Joins are good candidates, because join computation is expensive and join results are likely to be reused. We consider aggregation operators, because the result size of an aggregation is typically small while its input size is large. Thus, we can save large computational cost by paying a small storage and creation cost. Likewise, projections can also reduce the size of their input considerably. We do not consider selections as view candidates, because materializing the input of the selection and partitioning it on the attribute used in the selection is usually more effective than using selections along.

DEFINITION 6 (VIEW CANDIDATES). *For a query Q and view configuration* C*, the set* $\mathcal{V}_{cand}(Q)$ *of view candidates for* Q *contains all subqueries* Q' *of* Q *that fulfill the following conditions:*

- Q' *is of the form* $\gamma(Q_1)$, $Q_1 \bowtie Q_2$*, or* $\pi(Q_1)$
- Q' *does not exist in* $\mathcal{V}$

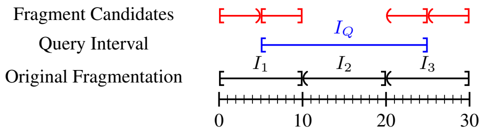
6.2 Partition Candidates

Similar to our view candidate generation approach, we want to use the characteristics of the current workload to guide the partition candidate generation. Note that we may maintain multiple partitions of the same view on different attributes. Given a current configuration of partitioned views C and statistics STAT kept for this configuration as well as for candidates, we consider new fragment candidates based on the selection conditions applied by a query. For every conjunction in the condition of a selection, i.e., a selection $\sigma_{l \leq A \leq u}(Q')$, which is a subquery of the current query Q, we consider new partition candidates for the view corresponding to Q', say V, based on the selection condition over attribute A. For the following discussions, without loss of generality, we assume $l \geq \underline{A}$ where $\underline{A}$ is the lowerbound of the domain of A, and $u \leq \bar{A}$ where $\bar{A}$ is the upperbound of the domain of A. It is trivial to replace l with $\underline{A}$ and similar for u when the above conditions do not hold. We have to distinguish several cases: 1) if we have not materialized Q' as a view V yet. In this case, we use l and u to split the potential fragments in $\mathcal{P}_{\textsc{Stat}}(V, A)$ which contain these points. If we have not yet gathered any intervals for this partition of V yet ($\mathcal{P}_{\textsc{Stat}}(V, A) = \emptyset$), then we initialize the partition with a single fragment: $\{\mathcal{D}(V, A)\}$ and then use l and u to split this fragment; 2) if a view V corresponding to Q' and a partition $\mathcal{P}(V, A)$ on attribute A already exists, then we again use the end points of the interval defined by the selection condition to consider splits of existing fragments that contain an end point as candidates. For each interval $I' = [l', u']$ of $\mathcal{P}(V, A)$ and the interval $I = [l, u]$, we create new candidates if either $l \in I'$ or $u \in I'$ using l respective u (or both) as split point(s).

DEFINITION 7 (PARTITION CANDIDATES). *Let Q be a query, C a view configuration, and* STAT *a view statistics. Consider a subquery* $\sigma_{l \leq A \leq u}(Q')$ *of* Q *where* Q' *corresponds to a view* V *in* $\mathcal{V}_{\textsc{Stat}}$ *and the intervals associated with partitioning* V *on attribute* A *(either* $\mathcal{P}(V, A)$ *if the view is in the pool or* $\mathcal{P}_{\textsc{Stat}}(V, A)$ *otherwise). We use* I *to denote* $[l, u]$*. For every interval* $I' = [l', u']$ *from* $\mathcal{P}(V, A)$ *respective* $\mathcal{P}_{\textsc{Stat}}(V, A)$ *we define the set of partition candidates* $\mathcal{P}_{cand}(V, A, Q')$ *according to* V*,* Q*, and* Q' *as the union of the sets of candidates for every such* I'*:*

1. *There is no overlap between these two intervals, i.e.,* $I' \cap I = \emptyset$*. In this case, no candidates are generated.*
2. *The query selection interval contains the partition interval, i.e.,* $I' \subseteq I$*. In this case, no candidates are generated.*
3. *The query selection interval overlaps the fragment interval from the left, i.e.,* $l < l' < u < u'$*. In this case, intervals* $[l', u]$ *and* $(u, u']$ *are considered as candidates.*
4. *The query selection interval overlaps the fragment interval from the right, i.e.,* $l' < l < u' < u$*. In this case, intervals* $[l', l)$ *and* $[l, u']$ *are considered as candidates.*
5. *The query selection interval is contained in the fragment interval, i.e.,* $I \subset I'$*. In this case, we consider three intervals as candidates:* $[l', l)$*,* $[l, u]$*, and* $(u, u']$*.*

EXAMPLE 3. *Consider a view* $V(A, B)$ *that is partitioned on attribute* A *using intervals* $I_1 = [0, 10]$*,* $I_2 = (10, 20]$ *and* $I_3 = (20, 30]$*. For an incoming query* $Q = \sigma_{5 \leq A \leq 25}(V)$ *we would consider the following candidates. Interval* $I = [5, 25]$ *overlaps with* I_1 *on the right (case 4). Thus, we create candidates* $[0, 5)$ *and* $[5 - 10]$*. No candidates are generated for* I_2 *(case 2). Finally,* I *overlaps with* I_3 *from the left (case 3) and we generate additional candidates* $(20, 25]$ *and* $(25, 30]$*.*



7. VIEWS AND PARTITION SELECTION

Our view and fragment selection method consists of two steps: 1) exclude candidates for which we have not gathered enough evidence of their effectiveness in improving the performance of the workload and 2) decide which candidates to materialize and which ones to evict to keep the pool size below the limit (S_{max}). The second step ranks views and fragments based on their *value* (Φ) as defined below. For each new fragment, we either create it by splitting existing fragments or create it as an overlapping fragment.

7.1 Cost and Benefit Model

We use a heuristic cost-benefit model to keep track of the "benefits" of view and fragment candidates. The benefits of a candidate are computed based on the potential savings in query execution time if this candidate would have been used to answer queries from the workload, the cost of creating it, its storage size, and other useful statistics for views and fragments. For candidates that have not been generated yet, we estimate their storage size and creation cost. We use this information to select which candidates to materialize and to decide which candidates to evict to make space for more competitive candidates.

View Statistics. For each view (candidate) V we keep the following statistics in $\mathcal{V}_{\text{STAT}}$: the **storage size** $S(V)$ occupied by the view, a set of **timestamps** $T(V)$ when the view was used to answer a query, and the **creation cost** of the view $\text{COST}(V)$ (which is initially estimated when we first see this view as a candidate). The creation cost is replaced with the actual cost once the first query containing the view as a subquery has been executed. The same applies to $S(V)$.

We compute the accumulated benefit $\mathcal{B}(V, t_{now})$ for a view at time t_{now} as follows. $\mathcal{B}(V, t_{now})$ is the cost we (could) have saved by using the view. The benefit is defined as

$$\mathcal{B}(V, t_{now}) = \sum_{Q \text{ used } V \text{ at } t} (\text{COST}(Q) - \text{COST}(Q/V)) \cdot \text{DEC}(t_{now}, t)$$

where $\text{COST}(Q)$ is the cost of query Q without using the view, $\text{COST}(Q/V)$ is the cost of running the query when using view V, and $\text{DEC}(t_{now}, t)$ is a monotonically decreasing function (in $t_{now} - t$) mapping the current time (t_{now}) and time when query Q was executed (t) to a value in $[0, 1]$. $\text{DEC}(t_{now}, t)$ is used to weight past cost savings by their age. This enables our approach to adapt to a changing workload. In our implementation we use the decay function as defined below which times out any benefit after a threshold t_{max} and otherwise counts it proportionally based on $\frac{t}{t_{now}}$.

$$\text{DEC}(t_{now}, t) = \begin{cases} 0 & \text{if } (t_{now} - t) > t_{max} \\ \frac{t}{t_{now}} & \text{otherwise} \end{cases}$$

View Value. Similar to Nectar [15], for each view V in the pool and candidate in $\mathcal{V}_{\text{STAT}}$ we compute its "value" at time t_{now} as a cost-benefit ratio $\Phi(V, t_{now})$. We use Φ during view selection to determine which views should be in the next configuration (views with a higher value are preferred over views of lower value). Using $\text{COST}(V)$, the accumulated benefit $\mathcal{B}(V, t_{now})$, and size $S(V)$, we define $\Phi(V, t_{now})$ as:

$$\Phi(V, t_{now}) = \frac{\text{COST}(V) \cdot \mathcal{B}(V, t_{now})}{S(V)}$$

The intuition behind the definition of $\Phi(V, t_{now})$ is that when a view is expensive to generate or the accumulated benefit of the view is large, its value is high and it is preferred over views with lower value. On the other hand, if the size of the materialized view is large, it is less competitive than other views of smaller size and similar benefits.

Fragment Statistics. Similar to view statistics we also keep separate statistics for every fragment in $\mathcal{P}(V, A)$ (a partition of view V on attribute A that is in the pool) as well as $\mathcal{P}_{\text{STAT}}(V, A)$ (a potential fragment candidate which is currently not materialized, but we have considered as a candidate before). For each such interval I (corresponding to a fragment F) we maintain the following information: the **storage size** of the fragment $S(I)$, its **creation cost** $\text{COST}(I)$, and a set of **timestamps** $T(I)$ when the fragment was hit (it was or could have been used to answer a query). These timestamps are used to compute the fragment value in a similar fashion as the view value explained above. We define the cost of creating the fragment to be the same as the cost of creating the partitioned view this fragment belongs to. This is because in order to recompute the fragment if it is not in the pool, we have to recompute the view's query and partition it.

Fragment Value. The value of a fragment is also modeled as a cost-benefit ratio in the same fashion as for views with the exception that benefits are computed as a ratio of the view creation cost and the relative size of the fragment compared to the total size of the view. The accumulated benefit for a fragment I is computed as

$$\mathcal{B}(I, t_{now}) = \sum_{Q \text{ used } I \text{ at } t} (\frac{S(I)}{S(V)} \cdot \text{COST}(V) \cdot \text{DEC}(t_{now}, t))$$

and

$$\Phi(I, t_{now}) = \frac{\text{COST}(V) \cdot \mathcal{B}(I, t_{now})}{S(I)}$$

Probabilistic Fragment Benefit Model. The definition of the value of a fragment above ignores the fact that fragments in a partition of a view do not exist independent of each other, i.e., two fragments may be "neighbors" (e.g., $[0, 10]$ and $[11, 30]$) or may be quite dissimilar (e.g., $[0, 10]$ and $[1000, 1010]$). If we treat the hits on fragments we have observed so far in the workload as samples of a probability distribution, then when using these samples to determine the underlying distribution it would be natural to consider "distance" between fragments in the mechanism that determines the distribution. For instance, if we observe a large number of hits on a fragment $[0, 5]$ and no hits on fragments $[6, 10]$ as well as $[11, 15]$, then it is still reasonable to assume that fragment $[6, 10]$ which is close to a "hot spot" has a higher likelihood to be used in the future than fragment $[11, 15]$. Based on this observation, we present a mechanism for adjusting the number of hits per fragment. Define the number of hits $H(I)$ for a fragment I as $H(I) = \sum_{Q \text{ used } I \text{ at } t} \text{DEC}(t_{now}, t)$).We now define the adjusted number of hits $H_A(I)$ to compute a more realistic fragment value.

Consider a partition $\mathbb{P}_{\mathcal{I}}(V.A)$ for a view V. Note that we do not require that all intervals in $\mathcal{I}$ are currently in the pool. We keep statistics for each $I \in \mathcal{I}$ no matter whether materialized or not. Let H_{total} denote the total hits over all fragments of $\mathcal{I}$ adjusted by our decay function, i.e., $H_{total} = \sum_{I \in \mathcal{I}} H(I)$. H_{total} is the total number of queries that used at least one fragment from $\mathbb{P}_{\mathcal{I}}(V.A)$ weighted by $\text{DEC}(t_{now}, t)$.

Based on the analysis of the real-life workloads presented in Section 1, it is reasonable to assume that a normal distribution underlies accesses to values of an attribute's domain. Thus, given the observed hits for fragments we want to choose the mean μ and variance σ^2 of a normal distribution such that the resulting distribution best fits the observed hits. Here we apply well-known techniques from statistics for computing the maximum likelihood estimators (MLE) for the mean $\hat{\mu}$ and variance $\hat{\sigma}^2$ of normal distributions [28] to do the curve fitting.

We split the domain of attribute A into equi-size intervals p_1, ..., p_n which we call parts to distinguish them from fragments. We choose a quantification such that no part p_i is partially contained in an interval $I \in \mathcal{I}$. For instance, for a domain $[0, 20]$ if $\mathcal{I} = \{[0, 10], [11, 15], [16, 20]\}$ we may choose parts of size 5: $\{[0, 5], [6, 10], [11, 15], [16, 20]\}$. Based on the hits recorded for fragments $I \in \mathcal{I}$ we then determine the hits for each part p_i. For each fragment, we split the number of hits to this fragment evenly to the parts that are contained in the fragment. Let $\mathcal{I}' \subseteq \mathcal{I}$ be the intervals containing p_i and $\#I$ the number of parts contained in interval I. We define $H(p_i) = \sum_{I \in \mathcal{I}'} \frac{H(I)}{\#I}$, i.e., summing up the number of hits for each interval containing the part weighted based on the number of parts the interval contains. The likelihood function L for a standard distribution $N(\mu, \sigma)$ and set of observations $\{p_1, \dots, p_n\}$ determines how likely it is that this particular distribution produced the given set of observations. It is defined as:

$$L(\mu, \sigma^2; p_1, p_2, ..., p_n) = (2\pi\sigma^2)^{-n/2} exp(-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(p_i - \mu)^2)$$

By solving the log-likelihood function of the above function we have the maximum likelihood estimator mean and variance:

$$\hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} p_i \qquad \hat{\sigma}_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}(p_i - \hat{\mu}_n)^2$$

The distribution $N(\hat{\mu}, \hat{\sigma})$ is the normal distribution which is most likely given the observations (it maximizes the likelihood function L). Note that we use the adjusted sample variance for the estimator $\hat{\sigma}_n^2$ because usually we do not expect a very large number of fragments for a view. This is a standard approach in statistics [28].

Note that since the MLE method is inexpensive we repeatedly adapt the estimation during the selection process for each incoming query. Based on the smoothed distribution of value accesses $N(\hat{\mu}, \hat{\sigma})$ we get from the maximum likelihood method and H_{total}, the total number of hits over all partitions, we compute the adjusted hits for a fragment $I = [l, u]$ as:

$$H_A(I) = H_{total} \cdot (P(x \leq u) - P(x \leq l))$$

Here $P(x \leq c)$ is an estimate (which ignores interval overlap) of how likely an access to a point in the interval $[-\infty, c]$ is computed over the normal distribution we have estimated using MLE. Note that this technique works for any probability distribution such as a Zipfian distribution or a mixture of distributions as long as it is feasible to compute the MLE of such a distribution given the observations. Here we choose the normal distribution, because it closely resembles the access patterns we have found in the real world workloads we have studied.

7.2 Filtering View and Partition Candidates

Our goal is to only save an intermediate result as a materialized view if this view is likely to be reused in the future and if the benefit of reuse $\mathcal{B}(V, t_{now})$ will offset the cost $\textsc{Cost}(V)$ of materializing this view. Thus, the subset of candidates we consider for materialization is:

$$\mathcal{V}_{sel} = \{V \mid V \in \mathcal{V}_{cand} \wedge \textsc{Cost}(V) \leq \mathcal{B}(V, t_{now})\}$$

We apply a similar filtering step for fragment candidates. This step is only applied for fragment candidates of existing partitions, i.e., when we decide whether to refine an existing partition based on selection, but not for candidates fragments for partitions which are not in the pool yet. Here we use the total benefits for a fragment computed based on its adjusted hits (using the estimated probability distribution of hits). Consider a candidate fragment I_{cand} for partition $\mathcal{P}(V, A)$ that is a candidate for the current query. The cost of creating I_{cand} depends on which fragments are currently in $\mathcal{P}(V, A)$. To materialize I_{cand} we have to read all fragments I such that $I \cap I_{cand} \neq \emptyset$, extract data that belongs to I_{cand} and then store I_{cand}. While we do not know upfront the actual size $S(I_{cand})$ for a fragment I_{cand}, we can estimate it based on the sizes of fragments currently in $\mathcal{P}(V, A)$ that overlap with I_{cand}. We assume that values are uniformly distributed within each fragment, and thus we can use the relative overlap between I_{cand} and an intervals in $\mathcal{P}(V, A)$ to estimate the size as:

$$S(I_{cand}) = \sum_{I \in \mathcal{P}(V,A): I \cap I_{cand} \neq \emptyset} \frac{\|I_{cand} \cap I\|}{\|I\|} \cdot S(I)$$

Based on this estimate of the size for a candidate fragment we have not materialized yet (otherwise we would know its size) we estimate the cost of creating the fragment as:

$$\textsc{Cost}(I_{cand}) = w_{write} \cdot S(I_{cand}) + \sum_{I \in \mathcal{P}(V,A): I \cap I_{cand} \neq \emptyset} w_{read} \cdot S(I)$$

Here w_{read} (and w_{write}) denote implementation specific constants for reading (respectively, writing) data. In our implementation of Deepsea, w_{write} is typically much larger than w_{read} if we store a fragment in HDFS. Given the cost and estimated size, we only consider fragments for which the benefits are larger than the creation cost:

$$\mathcal{P}_{sel} = \{I \mid I \in \mathcal{P}_{cand} \wedge \textsc{Cost}(I) \leq \mathcal{B}(I)\}$$

7.3 View and Fragment Selection

Given the prefiltered set of candidate views $\mathcal{V}_{sel}$, we now determine which of them to materialize (admit to the pool). In case this causes the total size of the views and fragments to exceed the limit S_{max}, we also have to decide which views or fragments to evict from the pool. Note that for selection we treat each fragment of a view independently. That is, the views in the pool do not partake in the selection process, only their fragments. However, candidate views and fragments are treated alike (candidate fragments are only created for partitioned views in the pool and view candidates are only created for views that do not currently exist in the pool). Thus, the set of views and fragments that are considered to be selected for the next configuration are:

$$\textsc{AllCand} = \mathcal{V}_{sel} \cup \mathcal{P}_{sel} \cup \bigcup_{V \in \mathcal{V}, A \in \textsc{Schema}(V)} \mathcal{P}(V, A)$$

We rank the elements (views and fragments) in this set based on their value Φ (defined in Section 7.1). We then greedily add elements to the new configuration based on their rank.

Let $\textsc{AllCand}[i]$ be the ith element from $\textsc{AllCand}$ according to Φ in decreasing order. We keep the first n elements from $\textsc{AllCand}$ for the largest n such that $\sum_{i=0}^{n} S(\textsc{AllCand}[i]) \leq S_{max}$:

$$C_{i+1} = \{\textsc{AllCand}[i] \mid i \in \{0, \dots, n\}\}$$

where

$$n = \underset{j \in \mathbb{N}}{\text{argmax}}(\sum_{i=0}^{j} S(\textsc{AllCand}[i]) \leq S_{max})$$

8. VIEW AND PARTITION MATCHING

The first important step when processing a query Q with our approach is to determine which views (whether in the pool or not) can be used to answer query Q. We call this process *matching*. The purpose of this step is to 1) update the statistics of views and fragments that could be used to answer query Q and 2) to determine the most efficient way of executing the query given the current configuration. The problem of finding all rewritings of a query Q given a set of views, i.e., queries that use the views and are equivalent to the input query, has often been called *query answering with views*. As mentioned earlier this problem in its full generality is undecidable for the class of queries we are interested in. We adopt a technique from Goldstein and Larson [14] that uses a sufficient condition to determine whether a view can be used to answer a query and indexes views such that this condition can be efficiently tested. We use a modified version of the index structure introduced in this work adapted for partitioned views to speed up matching.

8.1 A Sufficient Condition for Matching

The sufficient matching condition of Goldstein and Larson is checked over a representation of the query and the view (called *signature*) which is mostly independent of syntax, but can nonetheless be constructed from a concrete plan for the query. Signatures abstract away certain syntactic features such as join order. Our logical matching approach compares subqueries of a query with materialized views by computing the signatures for both the view and the subquery, and then checking the sufficient condition of Goldstein and Larson. Thus, we are able to also match parts of a query with a view. The signature of a query consists of the set of relations accessed by the query (*relation classes*), information on join and selection predicates (*attribute equivalence classes*, *selection predicate ranges*, and *remaining selection predicates*), projections, aggregation functions and group-by expressions. We refer the reader to Goldstein and Larson [14] for definitions of these abstractions.

8.2 Partition Matching

Once we have determined a rewriting using the views, the next step is to determine which partition of each view included in the rewriting to use and for each partition determine a subset of the fragments to be used. In order to match a fragment and a query, we must first find a match between the view represented by the fragment and the query. Note that a fragment of a view V corresponds to a view $\sigma_{l<A<u}(V)$ where A is the attribute on which V is partitioned on, and u and l are the boundaries of the fragment.

For every view V partitioned on A that is matched against a subquery Q' of the current query Q, we determine the restrictions Q' places on attribute A. This is done by using information about value ranges of selection conditions that are stored in the *Attribute Value Ranges* part of a query's signature (see [14] for a detailed explanation of the signature). Given our definitions of overlapping partitioning, the matching between a set of overlapped fragments and a query selection range is a set cover problem and thus is intractable. We use Algorithm 2 that greedily matches the fragments to a query selection range. Note that we use $\underline{I}$ to denote the lower and $\bar{I}$ to denote the upper bound of an interval I. We look for a set of fragments whose union covers the selection range. We maintain a variable $u_{covered}$ that stores the upper bound of the region covered so far. $u_{covered}$ is initialized to the lower bound of the selection range of the query u_θ. In each iteration of the loop, we greedily add the fragment that has the largest lower bound among the fragments that cover $u_{covered}$ from the left.

Algorithm 2 Partition Matching Algorithm

```
procedure PartitionMatching($\theta, \mathcal{I}$)
    $u_\theta \leftarrow$ Upperbound of $\theta$
    $l_\theta \leftarrow$ Lowerbound of $\theta$
    $F \leftarrow \emptyset$
    $u_{covered} \leftarrow l_\theta$
    while $u_{covered} < u_\theta$ do
        $\mathcal{I}_{cand} \leftarrow \{I \mid I \in \mathcal{I} \wedge \underline{I} \leq u_{covered} \wedge \bar{I} > u_{covered}\}$
        $I_{cur} = \text{argmax}_{I \in \mathcal{I}_{cand}} \underline{I}$
        $u_{covered} \leftarrow \bar{I}_{cur}$
        $F \leftarrow F \cup \{I_{cur}\}$
    end while
    return $F$
```

8.3 Indexing Partitioned Views

When computing matches between a query Q and a set of materialized views, it would be too slow to evaluate the sufficient matching condition over the signatures of all pairs of subqueries of Q and views in the pool. We adapt an in-memory index for view signatures called a filter tree [14] to be able to prune the search space early-on. A node in the tree is represented by a set of (key, pointer) pairs, where the key is a set of values, and the pointer points to a node on the next level. Each level represents one of the signature parts, e.g., the relations accessed by the view. The pointer of a leaf node points to a view. For each view, we store its partition information. For each partition of a view, we store the boundaries and statistics for each of its fragments. Note that we allow multiple partitions for the same view to exist as long as they are on different attributes. The search key for a query Q is its signature. We also use this index to keep the statistics for view and partition candidates (covered in Section 6).

8.4 Updating View and Partition Statistics

During view matching we update the statistics we keep for each view and its fragments, no matter whether the view or fragment is currently in the pool or not. For every rewriting $Q_{rewr} \in Rewr(Q)$ let V be a view that has been used in Q_{rewr} and for each such view $\mathcal{P}(Q_{rewr}, V, A)$ be the fragments of the partition of V on attribute A that are accessed by Q_{rewr}. We update the statistics for each such view and its fragments to reflect that it could be used to answer the query Q using the formulas presented in Section 7.1.

9. IMPLEMENTATION

DeepSea extends Hive [27], an SQL-on-Hadoop system [1]. While we have chosen Hive, because it is relatively mature, our techniques are applicable for any system that supports declarative querying ontop of shared-nothing dataflow systems.

Query Processing in DeepSea. Figure 4 shows how a query is processed by DeepSea. We use the parser and semantic analyzer of Hive to transform the input query into an abstract syntax tree (AST). The AST is translated into a directed acyclic graph (DAG) of operators (operator tree) and a task DAG (task tree) is generated from the operator DAG. Task DAGs assign operators to map and reduce phases. Our view matching module (see Section 8) rewrites the task DAG by replacing subqueries with references to materialized views or fragments. The rewritten DAG is then transformed into a DAG of MapReduce jobs to form a execution plan. We have implemented a *partition operator* that splits its input based on a list of fragment predicates which determine which input tuple belongs to which fragment. The output for each fragment is routed to a file sink operator that writes the fragment's content to a file.

Simulator. Testing view and fragment selection strategies requires extensive experiments over a large number of diverse workloads.

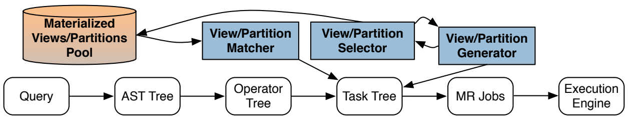


Figure 4: Query processing in DeepSea

Description	Values (default in bold)
Instance size	100GB, **500GB**
Pool size	50GB, 125GB, 250GB, 500GB, ∞
Query selectivity	**1% (Small)**, 5% (Medium), 25% (Big)
Query skew	Uniform (U), Light (L), **Heavy (H)**

Table 1: Parameters and their values

Since the benefits of partitioned views are more pronounced for large datasets, it is necessary to consider such datasets which results in large query runtimes. To be able to quickly test variations of a workload with different selection conditions ranges we have developed a simulator to study the efficiency of our selection algorithm and compare it to alternative approaches. We run a series of query templates with different selection patterns (introduced in Section 10) and gather statistics such as the storage size of views and fragments as well as the elapsed time. The simulator keeps track of the query template and the selection pattern that is running. It builds the necessary views and partitions based on the selection strategies and the size limit of the materialized view pool. Once sufficient statistics have been gathered for a query template, we estimate the runtime of future executions of a query template using linear regression.

Bounding Fragment Size. There are situations where bounding the size of a fragment (from above or below) may be beneficial. If the access patterns of queries are limited to a small subrange of the domain of an attribute, then our approach may create very large fragments for the parts of the domain that are accessed infrequently. In general it would be beneficial to split such large fragments, because the potential benefit of large fragments is small while the overhead of creating a few medium sized fragments instead is not very high. We approach this problem by limiting the maximal size of the fragments we create relative to the size of a view. We define a threshold ϕ for the relative size of a fragment. When we materialize and partition a view, we split every fragment that is larger than $\phi \times S(V)$ into smaller, equi-sized fragments. Big data systems are usually built on top of distributed file systems that favors large block sizes. For instance, HDFS has a default block size of 128 MB (or 64 MB depending on the version). We use the file system's block size as the lower bound for fragment size.

10. EVALUATION

We evaluate our system using the big data benchmark suite BigBench [13]. We demonstrate the overall performance of DeepSea using queries and data distributions that are modeled based on the SDSS workload [2]. This ensures that our evaluation considers important characteristics of real workloads. We also use BigBench to generate a set of synthetic workloads that are tailored to evaluate our major contributions: adaptive and progressive partitioning, exploitation of fragment correlations, and overlapping partitioning.

We generate instances of size 100GB and 500GB, both with uniform distribution, for the synthetic workloads. Table 1 shows parameters that we vary in the experiments as independent variables. The default value for each variable is shown in bold. We use the default value for the experiments unless otherwise mentioned. We consider three different query selectivities: *Small* (S) means that the selection condition returns 1% of the data; *Medium* (M) means that the selection condition returns 5%; and *Big* (B) means 25%. We use three different distributions for selection conditions of queries: uniform distributed (U), lightly skewed (L), and heavily skewed (H). *Uniform* means that for a fixed interval size, we pick a set of intervals such that the mid-point of the intervals is uniformly distributed. *Lightly skewed* means the mid-point of the selection intervals follows a normal distribution over the domain with a variance set to 7.5% of the domain. *Heavily skewed* also uses a normal distribution, but with the variance set to 0.25% of the domain.

Our evaluation is conducted on a cluster of 32 nodes. One node is a dedicated master node with 8 threads and 48GB memory. Each of the remaining 31 slave nodes has 6 threads, 12GB memory, and a 400GB disk. All results are based on the average of at least three runs, unless mentioned explicitly.

10.1 Workload for a Real-Life Application

We demonstrate two key properties of our system on a real-life application: 1) we compare the performance of DeepSea when there is no size limit for the materialization pool to Hive that does not uses materialization and NP, a materialization strategy that stores each view without partitioning them; 2) we compare the performance of DeepSea when there is a size limit for the pool to state-of-the-art view selection strategies such as the one of Nectar [15].

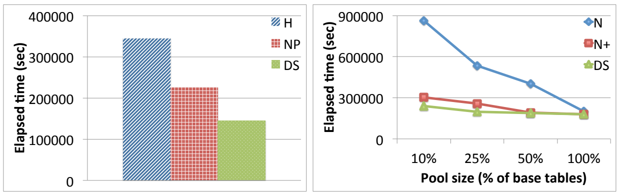
We create a histogram over the values of attribute `ra` for the table `PhotoPrimary` of SDSS. We then generate a BigBench dataset, and for all tables that contain attribute `item_sk` use the histogram that we obtained from SDSS attribute `ra` to sample values for `item_sk`. Furthermore, we generate a query workload: we pick ten query templates (Q1, Q5, Q7, Q9, Q12, Q16, Q20, Q26, Q29, Q30) from BigBench that contain joins, and we add a selection on attribute `item_sk` to these templates. We randomly pick 1000 selection ranges from the SDSS workload (selections on attribute `ra` of the table `PhotoPrimary`, kept in order of the query submission time). Next, we randomly picked a BigBench query template and mapped the selections of SDSS to selections on `item_sk` of the BigBench queries. Thus, we obtain a workload of 1000 BigBench queries simulating SDSS access patterns over an SDSS data distribution to evaluate the overall performance of DeepSea.

In this experiment, we compare DeepSea with two baselines. The first is the unmodified Hive system (*H* in the graphs). The second is a materialization strategy that does not use partitioning. We call this strategy non-partition (or *NP* in the graphs). This is akin to using a materialization strategy like ReStore [12]. However, in constrast to ReStore which only uses physical matching, *NP* applies our logical matching technique. Figure 5a shows performance results for the 500GB dataset without a pool size limit. Our approach requires only 64.2% of the time of non-partition materialization to execute the whole workload. Materialization without partitioning results in roughly 65.6% of the time of Vanilla Hive.

To evaluate the effectiveness of DeepSea's selection strategy, we compare it with the view selection strategy of Nectar [15]. Nectar does not consider accumulated benefit as a factor. To understand the performance gain due to the use of accumulated benefit in contrast with the other innovations in DeepSea, we extended Nectar's cost-benefit model to include the accumulated benefit of a view or fragment. The modified cost-benefit measure N^+ for views which we call *Nectar+* is: $N^+(V) = \frac{\textsc{Cost}(V) \times \mathcal{N}(V)}{S(V) \times \Delta T}$ where ΔT is the time elapsed since the last access to V and

$$\mathcal{N}(V) = \sum_{Q \text{ used } V \text{ at } t} (\textsc{Cost}(Q) - \textsc{Cost}(Q/V))$$

For fragments, we adapt our formula from Section 7.1 in a similar fashion by removing the application of the decay function. Fig-



(a) DS vs. NP vs. H (b) Selection strategies

Figure 5: Workload simulating SDSS (1000 queries), 500GB

ure 5b shows results for Nectar (N in the graph), Nectar+ (N+ in the graph), and DeepSea (DS in the graph) for different pool size limits. We observe that Nectar+ consistently outperforms Nectar, and DeepSea consistently outperforms Nectar+. When the size limit of the materialized view pool is relatively large (500GB, which is the total size of all base tables), the difference between Nectar, Nectar+ and DeepSea is marginal. When the size limit is shrunk to 10% of the total size of all base tables, DeepSea shows its strength requiring only ∼ 28% of the time of Nectar (20% faster than Nectar+) and being 30% faster than Vanilla Hive. DeepSea keeps fragments that can improve the overall performance in the pool, because they are neighbors of more frequently accessed fragments. Nectar and Nectar+, however, evict these fragments because of their low hit count. When the pool limit is decreased to 5% of the total database size, all three techniques perform poorly (worse than original Hive with no materialization (Figure 5a)). This is because with such a small materialized view pool, all three strategies evict fragments that are accessed earlier and admit fragments that are accessed more recently. Since evicted fragments may be accessed soon after eviction, there is an "oscillation" in the pool with extra working being done for the materialization and little or no gain seen from this extra work.

10.2 Adaptive and Progressive Partitioning

To understand the benefits of partitioning strategy, we compare DeepSea with equi-depth partitioning (E in the graphs or E followed by a number indicating the number of fragments). Equi-depth is a simple, non-adaptive and non-progressive alternative to DeepSea's partitioning approach. To evaluate the benefit of progressive partitioning standalone, we tease out the benefits of using DeepSea which is workload aware.

For this experiment, we do not bound the size of the largest fragment. We use instances of query template Q30 and vary the selection condition of this query to produce workload sequences where `Q30_i` denotes the i^{th} query in a sequence.

First we generate a sequence of queries that has small selectivity and is heavily skewed as defined at the beginning of this section. Figure 6 shows the cost of partitioned view creation and the cost for queries that reuse fragments. Figure 6a shows that when the number of generated fragments increases, the cost for creating and partitioning the view increases as well. In Figure 6b, we notice that if the same number of fragments are generated by both approaches (6 fragments in this experiment), equi-depth performs worse than DeepSea because of the larger size of fragments that must be read during query evaluation. Increasing the number of generated fragments for equi-depth reduces the average runtime for the following queries. However, when we set the number of fragments to be relatively large (60 fragments), performance decreases. Small fragment size affects performance negatively, because a large number of files has to be read and data is unevenly distributed among tasks. Figure 6c shows the cumulative time for the query sequence.

In addition to better performance, DeepSea also differs from equi-depth partitioning in terms of the execution of MapReduce jobs on the cluster. We analyze cluster utilization for the queries that reuse the generated fragments by running the default query sequence on the default dataset. Besides noticing the time needed in DeepSea is about 20% less than equi-depth, the number of map tasks issued to the Hadoop engine is about 40% to 50% more for equi-depth. The reason is that the fragments used by equi-depth to answer the query are larger than the ones used by DeepSea. Thus, the Hadoop engine issues more map tasks to parallelize the read as much as possible. This indicates that equi-depth uses more resources than DeepSea to answer the same queries.

We now investigate how characteristics of the workload affect the performance of DeepSea compared to non-adaptive partitioning approaches such as equi-depth. In addition to measuring time for running a workload of 10 such queries, we also project the time (using linear regression) for 100 queries. Figure 7 shows the performance of materialization without partitioning (NP), materialization using an equi-depth partition of a fixed size (E), and our DeepSea approach using workload-aware partitioning (DS) compared to Hive on a 500GB dataset. The settings are indicated by concatenating the abbreviations for the selectivity and query-skew settings, for example, ML stands for medium selectivity and a lightly skewed distribution over the selection ranges.

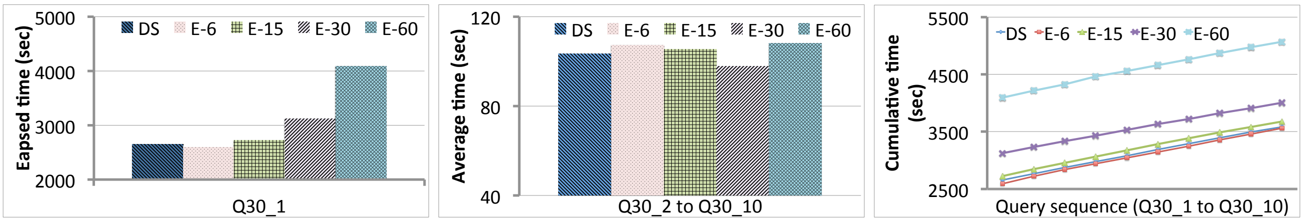
Figure 7a shows that both partitioning techniques (DeepSea and equi-depth) perform well compared to Hive and non-partition materialization in all experiments. When the selectivity is large, (indicated by B), our partition techniques can save 50 to 60% compared to Hive. For medium (M) selectivity, the partition techniques can save 60 to 70% and for small (S) selectivity the partition techniques can save 70 to 80%. Materialization alone without partitioning (NP) provides only a 15 to 25% improvement over Hive.

For uniformly distributed selections, DeepSea, as expected, does not provide a performance improvement over an equi-depth strategy (E). This is because equi-depth is tailored for such a distribution and the adaptive techniques of DeepSea do not pay off. However, for lightly skewed and heavily skewed selections, DeepSea has a noticeable advantage (up to 30%) over equi-depth partitioning. The performance of DeepSea increases and that of equi-depth decreases when introducing more skew (switching from uniformly distributed to lightly skewed and heavily skewed workloads). This is because we use the same number of fragments for DeepSea and equi-depth. When the workload is more and more skewed, there are fewer and smaller fragments needed by DeepSea to get the same benefit achieved by equi-depth.

Most optimizers will push down selections for reducing the size of intermediate results. Our materialization strategy requires that selections are not pushed down and hence we incur a performance hit initially. But even for small selectivities, this cost is quickly amortized over a workload. To understand when the additional work DeepSea does (by not pushing selections) is worth the cost, we plot the number of queries needed to recoup the cost of DeepSea in Figure 7b. Notice that for both DeepSea and equi-depth partitioning, the cost of not pushing a selection is recouped at almost the same point unless the workload is heavily skewed and includes queries with a large selectivity (requesting large portions of the data) in which case DeepSea has an advantage.

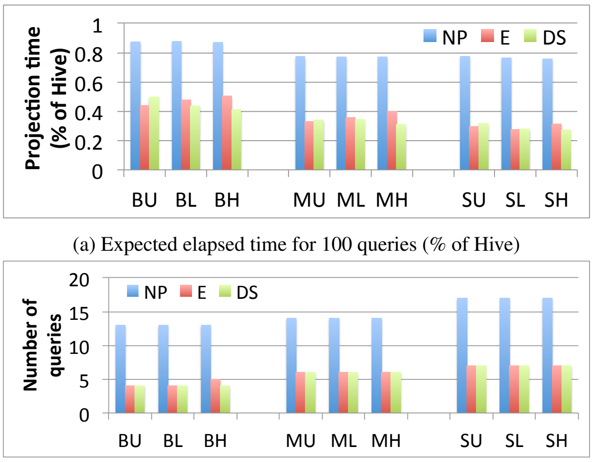
10.3 Exploitation of Fragment Correlations

We now compare our selection strategy that exploits fragment correlations against Nectar's strategy that is oblivious of such correlations. We use a workload that consists of ten queries (template Q30) that have big selectivity and are heavily skewed followed by another ten queries (also template Q30) that have small selectivity and are heavily skewed. We use a 500GB dataset with the pool



(a) Instrumented query materializing a view (b) Rewritten queries reusing a view (c) Cumulative time over the whole workload

Figure 6: Comparing equi-depth vs. adaptive partitioning (DeepSea) over workload using 10 instances of query template Q30, 100GB



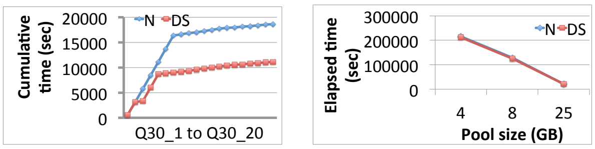
(a) Expected elapsed time for 100 queries (% of Hive)

(b) # of queries needed to recoup materialization cost

Figure 7: Varying selectivity and skew, Q30, 500GB

size limit set to 7GB. Figure 8a shows that DeepSea benefits from smoothing the distribution of hits to fragments from the same partition and, thus, is more likely to keep fragments that are similar to frequently accessed fragments.

Recall that we smoothen the distribution of hits over an attribute's range by fitting it to a normal distribution. Figure 8 shows how the performance of our approach is affected by the distribution underlying the selections in a workload. DeepSea significantly outperforms Nectar's selection strategy if the real hits follow a normal distribution. Importantly, it does not perform worse than Nectar if the selection ranges follow a radically different distribution (Zipf).



(a) Normal (b) Zipf

Figure 8: Selection ranges following Normal resp. Zipf distribution

10.4 Overlapping Partitioning

A key benefit of overlapping partitioning is that it writes less data when repartitioning for certain patterns that we observe in real-life applications frequently. In order to compare overlapping partitioning with horizontal partitioning, we generate a workload sequence of 30 queries from template Q30 with small selectivity and heavy skew. The selections of Q30_1 to Q30_10 have a midpoint of 20,000, the selections of Q30_11 to Q30_20 have a midpoint of 40,000, and the selections of Q30_21 to Q30_30 have a midpoint of 60,000. The domain of the selection attribute is [0, 400,000]. We generate this workload to simulate the common query selection pattern that we have observed in SDSS.

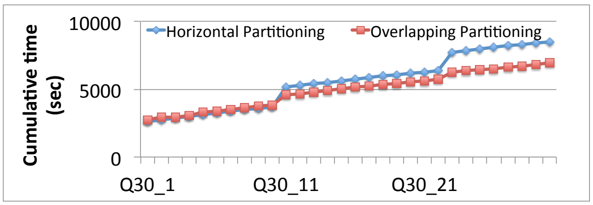
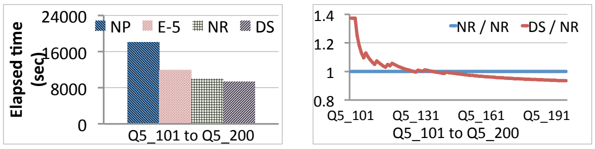


Figure 9: Overlapping partitioning (Q30_1 to Q30_30)

We are switching the pattern between Q30_10 and Q30_11 and between Q30_20 and Q30_21. Figure 9 shows that overlapping partitioning is more robust against changes in the workload, because it avoids writing a fragment that extends from the current upper bound of the selections to the upper bound of the domain that has not been queried yet.

We also generated a workload with 200 queries using query template Q5, all of which have big selectivity and are heavily skewed. The selection ranges for the first 100 queries were sampled from one distribution while the selection ranges for the next 100 queries follow a different distribution. Running this workload on the 100GB dataset, we compare against materialization without partitioning (NP in the graph), equi-depth partitioning with 5 fragments (E-5 in the graph) and DeepSea with no repartitioning (NR in the graph). Figure 10a shows for changing workloads, DeepSea outperforms the non-progressive approach that never repartitions by 7% and equi-depth partitioning by 27%. Figure 10b shows the cumulative time of DeepSea normalized to the cumulative time of the NR approach (no repartitioning), from query 101 (the first query following the new distribution) to query 200. DeepSea performs worse than NR for the first 30 queries because of the cost of repartitioning. This cost, however, is amortized by the subsequent queries.



(a) Cumulative time (b) Cumulative time ratio (DS/NR)

Figure 10: Adaptation to workload changes, Q5, 100GB

11. CONCLUSIONS

DeepSea is the first adaptive, progressive, workload-aware approach for automatic materialization and partitioning of views. Our cost-benefit model for both views and fragments takes the correlations among fragments into account. Our progressive partitioning accommodates both dynamic analytic workloads and exploratory workloads where users explore multiple regions in the data before finding (and then focusing on) a region of interest. DeepSea is implemented in Hive and our experiments demonstrate that our approach is more effective than traditional materialization techniques that do not consider the physical design of materialized views or do not adapt online to the workload. We also demonstrate that for real workloads, our view/fragment selection strategy outperforms state-of-the-art selection techniques when the materialized view pool has a small size limit.

In the short term, there are several interesting ways in which we can improve DeepSea including considering how to merge consecutive fragments that are mostly accessed together and how to best partition views on multiple attributes. DeepSea contributes to a rich literature on adaptive, progressive physical design strategies. For a fixed *memory overhead*, DeepSea selects a set of partitioned views and fragments of views to optimize the query performance (or minimize the *read overhead*). In the future, we would like to consider updates and explore how our techniques could be used with different optimization goals (including minimizing update overhead). We also would like to integrate our approach with query optimization, this would allow us to explore strategies that potentially select a more expensive query plan if it allows the materialization of interesting views that could benefit the workload.

12. ACKNOWLEDGMENTS

We thank Tilmann Rabl and Michael Frank for providing BigBench for our experiments. Stan Zdonik and Nesime Tatbul gave advice to improve the work. We also thank the anonymous reviewers for their helpful suggestions. This research was funded in part by a Bell Graduate Scholarship and NSERC.

13. REFERENCES

[1] Hadoop. http://hadoop.apache.org/.

[2] Sloan Digital Sky Survey. http://cas.sdss.org/.

[3] S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In *PODS*, pages 254–263, 1998.

[4] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, pages 496–505, 2000.

[5] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD*, pages 683–694, 2006.

[6] I. Alagiannis, D. Dash, K. Schnaitter, A. Ailamaki, and N. Polyzotis. An automated, yet interactive and portable DB designer. In *SIGMOD*, pages 1183–1186, 2010.

[7] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A Hands-free Adaptive Store. In *SIGMOD*, pages 1103–1114, 2014.

[8] E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *VLDB*, pages 156–165, 1997.

[9] N. Bruno and S. Chaudhuri. To tune or not to tune?: a lightweight physical design alerter. In *VLDB*, pages 499–510, 2006.

[10] S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. In *VLDB*, pages 3–14, 2007.

[11] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, M. Tan, et al. Semantic data caching and replacement. In *VLDB*, pages 330–341, 1996.

[12] I. Elghandour and A. Aboulnaga. ReStore: Reusing Results of MapReduce Jobs. *PVLDB*, 5(6):586–597, 2012.

[13] A. Ghazal, M. Hu, T. Rabl, F. Raab, M. Poess, A. Crolotte, and H. Jacobson. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In *SIGMOD*, pages 1197–1208, 2013.

[14] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: a practical, scalable solution. *SIGMOD Rec.*, 30(2):331–342, 2001.

[15] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: automatic management of data and computation in datacenters. In *OSDI*, pages 75–88, 2010.

[16] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[17] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 5(6):502–513, 2012.

[18] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[19] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves. An architecture for recycling intermediates in a column-store. *TODS*, 35(4):309–320, 2010.

[20] S. Jain, D. Moritz, B. Howe, and E. Lazowska. SQLShare: Results from a multi-year sql-as-a-service experiment. In *SIGMOD*, pages 281–293, 2016.

[21] A. Y. Levy, A. O. Mendelzon, and Y. Sagiv. Answering queries using views (extended abstract). In *PODS*, pages 95–104, 1995.

[22] F. Nagel, P. Boncz, and S. D. Viglas. Recycling in pipelined query evaluation. In *ICDE*, pages 338–349, 2013.

[23] S. Papadomanolakis and A. Ailamaki. AutoPart: Automating schema design for large scientific databases using data partitioning. In *SSDBM*, pages 383–392, 2004.

[24] L. Perez and C. Jermaine. History-aware query optimization with materialized intermediate views. In *ICDE*, pages 520–531, 2014.

[25] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *SIGMOD*, pages 558–569, 2002.

[26] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-line index selection for shifting workloads. In *ICDE*, pages 459–468, 2007.

[27] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.

[28] D. Wackerly, W. Mendenhall, and R. Scheaffer. *Mathematical Statistics with Applications*. Duxbury Press, 5th edition, 1996.

[29] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex sql queries using automatic summary tables. In *SIGMOD*, pages 105–116, 2000.

Matching Web Tables To DBpedia - A Feature Utility Study

Dominique Ritze, Christian Bizer
Data and Web Science Group, University of Mannheim,
B6, 26 68159 Mannheim, Germany
{dominique,chris}@informatik.uni-mannheim.de

ABSTRACT

Relational HTML tables on the Web contain data describing a multitude of entities and covering a wide range of topics. Thus, web tables are very useful for filling missing values in cross-domain knowledge bases such as DBpedia, YAGO, or the Google Knowledge Graph. Before web table data can be used to fill missing values, the tables need to be matched to the knowledge base in question. This involves three matching tasks: table-to-class matching, row-to-instance matching, and attribute-to-property matching. Various matching approaches have been proposed for each of these tasks. Unfortunately, the existing approaches are evaluated using different web table corpora. Each individual approach also only exploits a subset of the web table and knowledge base features that are potentially helpful for the matching tasks. These two shortcomings make it difficult to compare the different matching approaches and to judge the impact of each feature on the overall matching results.
This paper contributes to improve the understanding of the utility of different features for web table to knowledge base matching by reimplementing different matching techniques as well as similarity score aggregation methods from literature within a single matching framework and evaluating different combinations of these techniques against a single gold standard. The gold standard consists of class-, instance-, and property correspondences between the DBpedia knowledge base and web tables from the Web Data Commons web table corpus.

1. INTRODUCTION

Cross-domain knowledge bases such as DBpedia [18], YAGO [17], or the Google Knowledge Graph [36] are used as background knowledge within an increasing range of applications including web search, natural language understanding, data integration, and data mining. In order to realize their full potential within these applications, cross-domain knowledge bases need to be as complete, correct, and up-to-date as possible. One way to complement and keep a knowledge base up to date is to continuously integrate new knowledge from external sources into the knowledge base [10].

Relational HTML tables from the Web (also called web tables) are a useful source of external data for complementing and updating knowledge bases [31, 10, 40] as they cover a wide range of topics and contain a plethora of information. Before web table data can be used to fill missing values (”‘slot filling”’) or verify and update existing ones, the tables need to be matched to the knowledge base. This matching task can be divided into three subtasks: table-to-class matching, row-to-instance matching, and attribute-to-property matching. Beside the use case of complementing and updating knowledge bases, the matching of web tables is also necessary within other applications such as data search [40, 1] or table extension [41, 8, 21].

Matching web tables to knowledge bases is tricky as web tables are usually rather small with respect to their number of rows and attributes [19] and as for understanding the semantics of a table, it is often necessary to partly understand the content of the web page surrounding the table [41, 20]. Since everybody can put HTML tables on the Web, any kind of heterogeneity occurs within tables as well as on the web pages surrounding them. In order to deal with these issues, matching systems exploit different aspects of web tables (features) and also leverage the page content around the tables (context) [42, 41, 19].

There exists a decent body of research on web table to knowledge base matching [3, 22, 25, 39, 42, 16, 32]. Unfortunately, the existing methods often only consider a subset of the three matching subtasks and rely on a certain selection of web table and knowledge base features. In addition, it is quite difficult to compare evaluation results as the systems are tested using different web table corpora and different knowledge bases, which in some cases are also not publicly available. What is missing is an transparent experimental survey of the utility of the proposed matching features using a single public gold standard covering all three matching subtasks.

Whenever different features are used for matching, a method is required to combine the resulting similarity scores. While certain similarity aggregation methods work well for some tables, they might deliver bad results for other tables. Thus in addition to comparing different features and respective similarity functions, we also compare different similarity ag-

Series ISSN: 2367-2005

10.5441/002/edbt.2017.20

gregation methods. We focus the comparison on matrix prediction methods [33, 5] which are a specific type of similarity aggregation methods that predict the reliability of different features for each individual table and adapt the weights of the different features accordingly.

The contributions of this paper are twofold:

- We provide an overview and categorization of the web table and knowledge base features (together with respective similarity and similarity aggregation methods) that are used in state-of-the-art web table matching systems.
- We analyze the utility of the different matching features using a single, public gold standard that covers all three subtasks of the overall matching task. The gold standard consists of class-, instance-, and property correspondences between the DBpedia knowledge base [18] and web tables from the Web Data Commons table corpus [19].

The paper is organized as follows: Section 2 gives an overview of the overall matching process. Section 3 describes and categorizes the web table and knowledge base features. Section 4 discusses how the features can be used within the three matching tasks and describes the matchers that are employed to exploit the features within the experiments. The aggregation of similarity scores using matrix predictors is discussed in Section 5. Section 6 describes the gold standard that is used for the experiments. Section 7 compares the results of the different matrix prediction methods. Section 8 presents the matching results, compares them with existing results from the literature, and analyzes the utility of each feature for the matching tasks. Conclusions are drawn in Section 9.

2. OVERALL MATCHING PROCESS

We use the model and terminology introduced by Gal and Sagi in [15] to describe the overall process of matching a set of web tables and a knowledge base. Figure 1 shows an exemplary matching process. As input, two sources are required while as output, the process generates correspondences between manifestations of the sources. We consider everything within the sources a manifestation, e.g. manifestations are rows and columns of a table as well as instances, properties, and classes within a knowledge base. The internal components of a process are called first line matchers (1LM) and second line matchers (2LM).

A first line matcher (1LM) takes one feature of the manifestations as input and applies a similarity measure. As an example, a first line matcher gets the labels of the different attributes (columns) of a web table and the labels of the properties of a specific class within the knowledge base as feature, tokenizes both labels, removes stop words, and compares the resulting sets using the Jaccard similarity. The resulting similarity scores are stored as elements in a similarity matrix. In most cases, only considering a single feature is not sufficient for matching two sources. Thus, an ensemble of first line matchers is applied, ideally covering a wide variety of features exploiting different aspects of the web tables and the knowledge base.

Second line matchers (2LM) transform one or more similarity matrices into a resulting similarity matrix. Gal [14] distinguishes decisive and non-decisive second line matchers. Non-decisive matchers do not take any decision about the resulting correspondences, e.g. they only aggregate matrices. Typical aggregation strategies of non-decisive second line matchers are to take the maximal elements that can be found among the matrices or to weight each matrix and calculate a weighted sum. In the example depicted in Figure 1, the aggregation is performed by summing up the elements of both matrices. Non-decisive second line matchers are also referred to as combination methods [9] or matcher composition [11].

In contrast to non-decisive matchers, decisive second line matchers create correspondences between manifestations. For instance, a second-line matcher that applies a threshold is decisive because it excludes all pairs of manifestations having a similarity score below this threshold. It is often desirable that a single manifestation within a web table is only matched to a single manifestation in the knowledge base. To ensure this, so called 1 : 1 decisive second line matchers are used. In our example, the 1 : 1 matcher decides for the highest element within each matrix row and sets them to 1, all other elements are set to 0.

3. FEATURES

Features are different aspects of web tables and the knowledge base that serve as input for first line matchers. We perceive web tables as simple entity-attribute tables, meaning that each table describes a set of entities (rows in web tables) having a set of attributes (columns). For each entity-attribute pair, we can find the according value in a cell. We require every table to have an attribute that contains natural

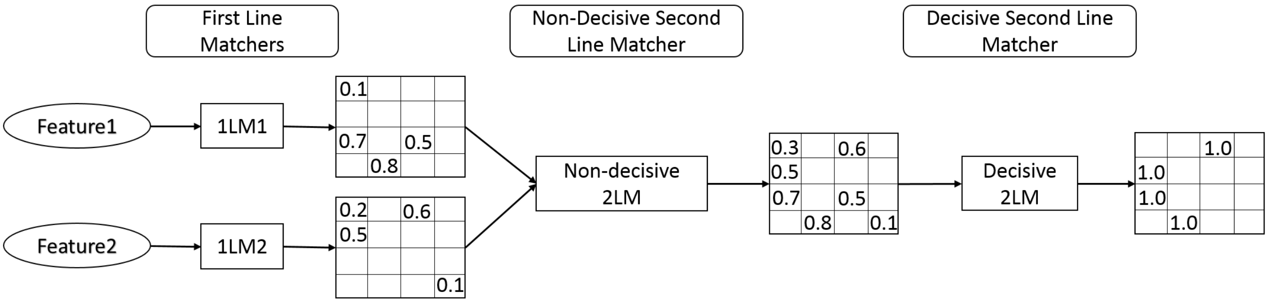


Figure 1: The matching process

language labels of the entities (called entity label attribute), e.g. the entity label of the city Mannheim is “Mannheim”. All other attributes are either of data type string, numeric or date. We currently do not consider any other data types like geographical coordinates as for instance taken into account by Cruz et al. [6] or tables with compound entity label attributes [20]. Further, each attribute is assumed to have a header (attribute label) which is some surface form of the attribute’s semantic intention. In order to distinguish between web tables and the knowledge base, we use the terms entity and attribute when talking about web tables and instance and property when talking about the knowledge base.

We use the categorization schema shown in Figure 2 for categorizing web tables features. In general, a feature can either be found in the table itself (Table T) or outside the table (Context C). As context features, we consider everything that is not directly contained in the table, e.g. the words surrounding the table. Context features can either be page attributes (CPA) like the page title or free text (CFT). We further divide table features into single features (TS), e.g. a label of an entity, and multiple features (TM), e.g. the set of all attribute labels occurring in a table. Single features refer to a value in a single cell while multiple features combine values coming from more than one cell.

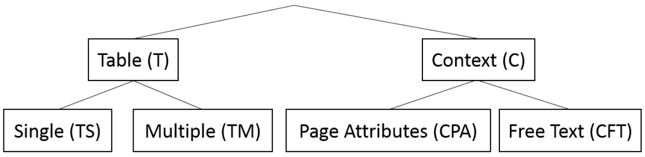


Figure 2: Web table feature categories

Table 1 gives an overview of all features that we consider and classifies them by category. As single table features we use the entity label, the attribute label, as well as the values that can be found in the cells. Multiple features are the entities as a whole, the set of attribute labels, and the table as text. We represent multiple features as bag-of-words. For example, the set of attribute labels can be characteristic for a table, e.g. the attribute labels “population” and “currency” give an important hint that the table describes different countries.

As context features, we use the page attributes title and URL and as free text feature the words surrounding the table. Often, the URL as well as the title of the web page contains information about the content of the table, e.g. the URL http://airportcodes.me/us-airport-codes indicates that a table found on this page might describe a set of airports. Context features are often not directly related to a specific table which makes it tricky to exploit them for matching. Nevertheless Yakout et al. [41] as well as Lehmberg [20] found context features to be crucial for high quality matching. Braunschweig et al. [1] take the surrounding words to extract attribute-specific context in order to find alternative names for attribute labels. The CONTEXT operator of the Octopus system [3] uses context features to find hidden attributes which are not explicitly described in the table.

Most state-of-the-art matching systems only exploit single table features [26, 43, 38, 40, 25, 22, 24, 39, 23, 16]. Multiple table features are considered by Wang et al. [40] (set of attribute labels), by the TableMiner system [42] (set of attribute labels and entities), and by the InfoGather system [41] (set of attribute labels, entities, and tables). Only the systems InfoGather and TableMiner leverage context features.

Table 2 shows the features of the knowledge base (DBpedia) that we exploit within the experiments. Analog to the table features, DBpedia features can either refer to a single triple, e.g. a triple representing the information about an instance label, or to a set of triples like the set of all abstracts of instances belonging to a certain class.

In addition to the web table and knowledge base features, external resource can be exploited for matching, e.g. general lexical databases like WordNet [12]. For matching web tables, systems use external resources which have been created based on co-occurrences [39], that leverage a web text corpus and natural language patterns to find relations between entities [35], or that exploit the anchor text of hyperlinks in order to find alternative surface forms of entity names [2].

4. MATCHING TASKS

The overall task of matching web tables against a knowledge base can be decomposed into the subtasks table-to-class matching, row-to-instance matching, and attribute-to-property matching. In this section, we provide an overview of the existing work on each subtask. Afterward, we describe the matching techniques that we have selected from the literature for our experiments. We employ the $T2KMatch$ matching framework [32][1] for the experiments and implement the selected techniques as first line matchers within

[1] http://dws.informatik.uni-mannheim.de/en/research/T2K

Table 1: Web table features

Feature	Description	Category
Entity label	The label of an entity	TS
Attribute label	The header of an attribute	TS
Value	The value that can be found in a cell	TS
Entity	The entity in one row represented as a bag-of-words	TM
Set of attribute labels	The set of all attribute labels in the table	TM
Table	The text of the table content without considering any structure	TM
URL	The URL of the web page from which the table has been extracted	CPA
Page title	The title of the web page	CPA
Surrounding words	The 200 words before and after the table	CFT

Table 2: DBpedia features

Feature	Description
Instance label	The name of the instance mentioned in the rdfs:label
Property label	The name of the property mentioned in the rdfs:label
Class label	The name of the class mentioned in the rdfs:label
Value	The literal or object that can be found in the object position of triples
Instance count	The number of times an instance is linked in the wikipedia corpus
Instance abstract	The DBpedia abstract describing an instance
Instance classes	The DBpedia classes (including the superclasses) to which an instance belongs to
Set of class instances	The set of instances belonging to a class
Set of class abstracts	The set of all abstracts of instances belonging to a class

the framework. The framework covers all three matching subtasks. Similar to PARIS [37], $T2KMatch$ iterates between instance- and schema matching until the similarity scores stabilize. Correspondences between tables and classes are chosen based on the initial results of the instance matching. Due to this decision, only instances of this class as well as properties defined for this class are taken into account. Thus, the class decision can have a strong influence on the other matching tasks [32].

4.1 Row-To-Instance Matching

The goal of row-to-instance matching is to find correspondences between instances in the knowledge base and entities described by individual rows of a web table. The row-to-instance matching task is tackled frequently by various systems in literature. Some systems purely rely on the label of the entity [26, 43] or on the label enriched with alternatives surface forms [22]. In addition, other systems also take the cell values into account [42, 38, 40, 25]. Most of them have in common that they query APIs to find potential instances, e.g. the Probase, Freebase or Wikiontology API. As a result, a ranked list of possible instances per entity is returned. The ranking function is not always known in detail but often relies on the popularity of an instance. Besides the internal API ranking, other rankings like the page rank of the according Wikipedia page of an instances can be added [26, 38]. As another source of information, Zhang [42] introduced context features (page title, surrounding words).

Within our experiments, we evaluate the utility of all single table features as well as the entity feature for the row-to-instance matching task. For this, we have implemented the following first line matchers within the $T2KMatch$ framework:

Entity Label Matcher: Before the entity label can be matched, we need to identify the attribute of the web tables that contains the entity label (entity label attribute). For determining the entity label attribute, we use a heuristic which exploits the uniqueness of the attribute values and falls back to the order of the attributes for breaking ties [32]. For matching the entity label, we apply the entity label matcher that is included in $T2K$. The matcher compares the entity label with the instance label using a generalized Jaccard with Levenshtein as inner measure. Only the top 20 instances with respect to the similarities are considered further for each entity.

Value-based Entity Matcher: $T2KMatch$ implements a value matcher which applies data type specific similarity measures. For strings, a generalized Jaccard with Levenshtein as inner measure, for numeric the deviation similarity introduced by Rinser et al. [30], and for dates a weighted date similarity is used which emphasizes the year over the month and day. The value similarities are weighted with the available attribute similarities and are aggregated per entity. If we already know that an attribute corresponds to a property, the similarities of the according values get a higher weight.

Surface Form Matcher: Web tables often use synonymous names ("'surface forms"') to refer to a single instance in the knowledge base, which is difficult to spot for pure string similarity measures. In order to be able to understand alternative names, we use a surface form catalog that has been created from anchor-texts of intra-Wikipedia links, Wikipedia article titles, and disambiguation pages [2]. Within the catalog, a TF-IDF score [34] is assigned to each surface form. We build a set of terms for each label resp. string value consisting of the label/value itself together with according surface forms. We add the three surface forms with the highest scores if the difference of the scores between the two best surface forms is smaller than 80%, otherwise we only add the surface form with the highest score. For each entity label resp. value, we build a set of terms containing the label or value as well as the alternative names. Each term in the set is compared using the entity label resp. value-based entity matcher and the maximal similarity per set is taken.

Popularity-based Matcher: The popularity-based matcher takes into account how popular an instance in the knowledge base is. For example, an instance with the label "Paris" can either refer to the capital of France or to the city in Texas. Both instances are equal regarding the label but most of the times, the city in France will be meant. To compute the popularity of an instance, we count the number of links in Wikipedia that point at the Wikipedia page which corresponds to the instance [7]. Similar methods based on the Wikipedias instance's page rank are applied by Mulwad et al. [26] and Syed at al. [38].

Abstract Matcher: Comparing the entity label and the values can be insufficient if the labels differ too much or if not all information about an instance is covered in the values, e.g. the capital of a country is not contained in the knowledge base as a value but stated in the abstract of the instance. This is especially relevant when thinking about the use case of filling missing values in the knowledge base. Therefore, the abstract matcher compares the entity as a whole with the abstracts of the instances, both represented as bag-of-words. For each entity represented as bag-of-words, we create a TF-IDF vector and compare it to the TF-IDF vectors constructed from the abstracts where at least one term overlaps. As similarity measure we use a combination of the denormalized cosine similarity (dot product) and Jaccard to prefer vectors that contain several different terms in contrast to vectors that cover only one term but this several times:

$A \bullet B + 1 - (\frac{1}{\|A \cap B\|})$ where A and B are TF-IDF vectors.

4.2 Attribute-To-Property Matching

The attribute-to-property matching task has the goal to assign properties from the knowledge base (both data type and object properties) to the attributes found in web tables. Existing attribute-to-property matching methods often focus on the matching only object properties to attributes[26, 25, 24], also named "relation discovery". As cross-domain knowledge bases usually contain data type and object properties, the goal in this paper is to detect correspondences for both types of properties. Beside exploiting attribute and property values, other methods also take the attribute label into account and compare it to the label of the property [22]. Similar to the instance matching task, the label comparison can be enhanced by including alternative attribute labels, e.g. computed based on co-occurrences [41]. The system introduced by Braunschweig et al. [1] discovers synonymous labels by using the context as well as the lexical database WordNet. Neumaier et al. [27] present a matching approach that explicitly focuses on numeric data which is published via open data portals.

Within our experiments, we evaluate solely single features for attribute-to-property matching and have implemented the following matchers for this:

Attribute Label Matcher: The attribute label can give hints which information is described by the attribute. For example, the label "capital" in a table about countries directly tells us that a property named "capital" is a better candidate than the property "largestCity" although the similarities of the values are very close. We use a generalized Jaccard with Levenshtein as inner measure to compare the attribute and property label.

WordNet Matcher: To solve alternative names for attribute labels, we consult the lexical database WordNet which has also been used by Braunschweig et al. [1]. WordNet is frequently applied in various research areas, e.g. in the field of ontology matching. Besides synonyms, we take hypernyms and hyponyms (also inherited, maximal five, only coming from the first synset) into account. As an example, for the attribute label "country" the terms "state", "nation", "land" and "commonwealth" can be found in WordNet. We again apply a set-based comparison which returns the maximal similarity scores.

Dictionary Matcher: While WordNet is a general source of information, we additionally create a dictionary for attribute labels based on the results of matching the Web Data Commons Web Tables Corpus to DBpedia with $T2KMatch$. As a result, from 33 million tables around 1 million tables have at least one instance correspondence to DBpedia [31]. We group the property correspondences and extract the according labels of the attributes that have been matched to a property. Thus, we are able to generate a dictionary containing the property label together with the attribute labels that, based on the matching, seem to be synonymous. At this point, the dictionary includes a lot of noise, e.g. the term "name" is a synonym for almost every property. A filtering based on the number of occurrences or on the number of web sites is not useful, since the rare cases are most promising. Thus, we apply a filter which excludes all attribute labels that are assigned to more than 20 different properties because they do not provide any benefit. The comparison is the same as for the other matchers including external resources. A related approach is performed by Yakout et al. [41] where synonyms of attribute labels are generated based on web tables that have been matched among each other.

Duplicate-based Attribute Matcher: The duplicate-based attribute matcher is the counterpart of the value-based entity matcher: The computed value similarities are weighted with the according instance similarities and are aggregated over the attribute. Thus, if two values are similar and the associated entity instance pair is similar, it has a positive influence on the similarity of the attribute property pair, see [32] for more details.

4.3 Table-To-Class Matching

The goal of table-to-class matching is to assign the class from the knowledge base to a web table which fits best to the content of the whole table. Assigning the class to which the majority of the instances in the table belong to is most common strategy for table-to-class matching [22, 42, 39, 23, 38, 43, 16]. On top of this approach, methods take also the specificity of a class into account [25], exploit the set of attribute labels [40] or consider the context [42].

We evaluate the utility of features from the categories "table multiple" and "context" for table-to-class matching and have implemented the following matchers for this:

Page Attribute Matcher: We process the page attributes page title and URL by applying stop word removal and simple stemming. The similarity of a page attribute to a class of the knowledge base is the number of characters of the class label normalized by the number of characters in the page attribute.

Text Matcher: Ideally, the set of abstracts belonging to instances of a class contains not only the instance labels and associated property labels but also significant clue words. We use this matcher for the features "set of attribute labels", "table" and "surrounding words". All features are represented as bag-of-words. After removing stop words, we build TF-IDF vectors indicating the characteristic terms of the table and the classes. We apply the same similarity measure which is used by the abstract matcher.

Majority-based Matcher: Based on the initial similarities of entities to instances computed by the entity label matcher, we take the classes of the instances and count how often they occur. If an instance belongs to more than one class, the instance counts for all of them. Such a matching approach has for example been applied by Limaye et al. [22] to assign classes to attributes covering named entities.

Frequency-based Matcher: Ideally, we want to find correspondences to specific classes over general classes which is not captured by the majority-based class matcher. Similar to Mulwad et al. [25], we define the specificity of a class as following:

$$spec(c) = 1 - \frac{\|c\|}{\max_{d \in C} \|d\|}$$

where c represents a particular class and C the set of all classes in DBpedia.

Agreement Matcher: The agreement matcher is a second line matcher which exploits the amount of class matchers operating on features covering different aspects. Although the matchers might not agree on the best class to choose, a class which is found by all the matchers is usually a good candidate. We propose the agreement matcher which takes the results of all other class matchers and counts how often they agree per class. In this case, all classes are counted having a similarity score greater than zero.

The results of our matching experiments are presented in Section 8.

5. SIMILARITY SCORE AGGREGATION

Each of the previously described matchers generates a similarity matrix as result. Depending on the task, these matrices contain the similarities between the entities and instances, attributes and properties or the table and classes. In order to generate the correspondence, all matrices dealing with the same task need to be combined which is the task of a non-decisive second line matcher. Most approaches in the field of web table matching use a weighted aggregation to combine similarity matrices. While some of them empirically determine the weights, e.g. TableMiner [42], others employ machine learning to find appropriate weights [41, 22]. All existing approaches for web table matching have in common that they use the same weights for all tables. Due to the diversity of tables, one single set of weights might not be the best solution. To overcome this issue, we use a quality-driven combination strategy which adapts itself for each individual table. Such strategies have been shown as promising in the field of ontology matching [5]. The approach tries to measure the reliability of matchers by applying so called matrix predictors [33] on the generated similarity matrices. The predicted reliability is then used as weight for each matrix. Since the prediction is individually performed on each matrix, the reliability of a matcher can differ for each table and in turn we are able to use weights which are tailored to a table.

We evaluate three different matrix predictors: the average predictor (P_{avg}), standard deviation predictor (P_{stdev}) [33] as well as a predictor (P_{herf}) which bases on the Herfindahl Index [29] and estimates the diversity of a matrix.

Average: Based on the assumption that a high element in the similarity matrix leads to a correct correspondence, a matrix with many high elements is preferred over a matrix with less high elements. We compute the average of a matrix M as following:

$$P_{avg}(M) = \frac{\sum_{i,j|e_{i,j}>0} e_{i,j}}{\sum_{i,j|e_{i,j}>0} 1}$$

Standard Deviation: In addition to the average, the standard deviation indicates whether the elements in the matrix are all close to the average. Formally:

$$P_{stdev}(M) = \sqrt{\frac{\sum_{i,j|e_{i,j}>0}(e_{i,j}-\mu)^2}{N}}$$

μ is the average and N is the number of non-zero elements.

Normalized Herfindahl Index: The Herfindahl Index (HHI) [29] is an economic concept which measures the size of firms in relation to the industry and serves as an indicator of the amount of competition among them. A high Herfindahl Index indicates that one firm has a monopoly while a low Herfindahl Index indicates a lot of competition. We use this concept to determine the diversity of each matrix row and in turn of the matrix itself. Our matrix predictor based on the Herfindahl Index is similar to the recently proposed predictor *Match Competitor Deviation* [13] which compares the elements of each matrix row with its average.

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

Figure 3: Matrix row with the highest HHI (1.0)

$$\begin{bmatrix} 0.1 & 0.1 & 0.1 & 0.1 \end{bmatrix}$$

Figure 4: Matrix row with the lowest HHI (0.25)

Figure 3 and Figure 4 show the highest and lowest possible case for a four-dimensional matrix row. At best, we find exactly one element larger than zero while all other elements are zero. Having this ideal case, we can perfectly see which pair fits. In contrast, a matrix row which has exactly the same element for each pair does not help at all to decide for correspondences. We compute the normalized Herfindahl Index for each matrix row which ranges between $1/n$ and 1.0 where n is the dimension of the matrix row. That is the reason why the matrix row in Figure 3 has a normalized Herfindahl Index of 1.0 and the matrix row in Figure 4 of 0.25. To get an estimation per matrix, we build the sum over all Herfindahl Indices per matrix row and normalize it. Formally:

$$P_{herf}(M) = \frac{1}{V} \sum_i \frac{\sum_j e_{i,j}{}^2}{(\sum_j e_{i,j})^2}$$

where V represents the number of matrix rows in the matrix.

Section 7 presents the evaluation results of the different matrix predictors and determines the predictor that is most suitable for each matching task. Further, we discuss how the weights are distributed across the different matrices generated by the matchers.

6. GOLD STANDARD

We use Version 2 of the *T2D* entity-level gold standard[2] for our experiments. The gold standard consists of web tables from the Web Data Commons table corpus [19] which

[2] http://webdatacommons.org/webtables/goldstandardV2.html

has been extracted from the CommonCrawl web corpus[3]. During the extraction, the web tables are classified as layout, entity, relational, matrix and other tables. For the use case of filling missing values in a knowledge base, relational tables are most valuable as they contain relational data describing entities. However, as shown by Cafarella et al. [4], the vast majority of tables found in the Web are layout tables. In addition we have shown in [31] that only a very small fraction of the relational tables can actually be matched to the DBpedia knowledge base. Thus, it is important for a matching algorithm to be good at recognizing non-matching tables. For a gold standard, it is in turn important to contain non-matching tables.

Version 2 of the $T2D$ entity-level gold standard consists of row-to-instance, attribute-to-property, table-to-class correspondences between 779 web tables and the DBpedia knowledge base. The correspondences were created manually. In order cover the challenges that a web table matching system needs to face, the gold standard contains three types of tables: non-relational tables (layout, matrix, entity, other), relational tables that do not share any instance with DBpedia and relational tables for which least one instance correspondence can be found. Out of the 779 tables in the gold standard, 237 tables share at least one instance with DBpedia. The tables cover different topics including places, works, and people. Altogether, the gold standard contains 25 119 instance and 618 property correspondences. About half the property correspondences refer to entity label attributes, while 381 correspondences refer to other attributes (object as well as data type attributes). Detailed statistics about the gold standard are found on the web page mentioned above.

A major difference between Version 2 of the $T2D$ gold standard and the Limaye112 gold standard [22] is that the $T2D$ gold standard includes tables that cannot be matched to the knowledge base and thus forces matching systems to decide whether a table can be matched or not.

7. SIMILARITY AGGREGATION RESULTS

This section describes the experiments we perform regarding the similarity score aggregation using matrix predictors. Following Sagi and Gal [33], we measure the quality of a matrix predictor using the Pearson product-moment correlation coefficient [28]. With a correlation analysis, we can ensure that the weights chosen for the aggregation are well suitable. We perform the correlation analysis for the three matching tasks with the three introduced matrix preditors P_{avg}, P_{stdev} and P_{herf} on the evaluation measures precision P and recall R.

$$P = \frac{TP}{TP + FP} \qquad R = \frac{TP}{TP + FN}$$

While TP refers to the number of true positives, FP represents the number false positives and FN the number of false negatives.

If a predictor has a high correlation to precision respect recall and we use the prediction for weighting the similarity matrix, we assume that the result also has an according precision/recall.

Table 3 shows the results of the correlation analysis for the property and instance similarity matrices regarding precision, e.g. PP_{stdev}, and recall, e.g. RP_{stdev}. All predictor correlations are significant according to a two-sample paired t-test with significance level $\alpha = 0.001$. The analysis of the class similarity matrix predictors is not shown since the correlations are not significant. This results from the fact that only 237 tables in the gold standard can be assigned to a DBpedia class and in turn only for those tables we can compute a correlation with precision and recall. However, in practice the predictor P_{herf} shows the most promising results. The same holds for instance similarity matrices where P_{herf} has the highest correlation with precision as well as recall. In contrast, for property similarity matrices, P_{avg} correlates most. One reason is the comparably low amount of properties that can potentially be mapped to one attribute. Within a single matching task, the choice of the best performing predictor is in most cases consistent. One exception is the correlation of P_{herf} to the recall of the matrix generated by the popularity-based matcher since the most popular instances do not necessarily need to be the correct candidates. Based on the results, we use the prediction computed by P_{herf} as weights for the instance as well as for class similarity matrices and P_{avg} for the property similarity matrices in the matching experiments that we report in the next section.

Figure 5 shows the variations of weights for the similarity matrices coming from different matchers. We can see that median of the weights differ for the various matchers which in turn indicates the overall importance of the features across all tables for a certain matching task. For the instance matching task, the popularity of an instance seems to play a crucial role, followed by the label. Contrary, the values build the foundation for the property matching task. The size of the class as well as the amount of instance candidates belonging to one class, used in the frequency-based resp. majority-based matcher, forms the basis of the class matching task. Adding external resources like Wordnet only leads to slight changes of the weights.

Besides the median, the variations of the weights show that the actual utility of a feature depends on the individual matrix and in turn on the table. This supports our assumption that taking the same aggregation weights for all the tables is not always the best strategy. While the weight variations are very large for all matchers operating on attribute labels (attribute label-, wordnet- and dictionary matcher), this is the opposite for the matchers dealing with bag-of-words. A large variation implies that the reliability is predicted differently for various tables which in turn indicates that the attribute label is a suitable feature for some but not for all of the tables. This finding is reasonable since tables can either have attribute labels that perfectly fit to a property label like “capital” while others do not use any meaningful labels. For the bag-of-words matchers, the reliability is estimated quite similar but low for all the tables. Since they compare bag-of-words, they will always find a large amount of candidates.

8. MATCHING RESULTS

In this section, we report the results of our matching experiments and compare them to results from the literature. After applying the different matchers and aggregating their

[3] http://commoncrawl.org/

Table 3: Correlation of matrix predictors to precision and recall

First Line Matcher	PP_{stdev}	RP_{stdev}	PP_{avg}	RP_{avg}	PP_{herf}	RP_{herf}
	Property Similarity Matrices					
Attribute label matcher	**0.474**	0.415	0.433	**0.448**	0.215	0.209
Duplicate-based attribute matcher	0.048	0.094	**0.086**	**0.106**	-0.074	0.042
WordNet matcher	**0.425**	0.341	0.317	**0.367**	0.120	0.178
Dictionary matcher	0.360	0.274	**0.364**	**0.447**	0.130	0.150
mean	**0.327**	0.281	0.300	**0.342**	0.098	0.145
	Instance Similarity Matrices					
Entity label matcher	-0.167	0.092	-0.160	0.049	**0.233**	**0.232**
Value-based entity matcher	0.361	0.496	0.122	0.311	**0.378**	**0.531**
Surface form matcher	-0.291	-0.094	-0.294	-0.128	**0.241**	**0.238**
Popularity-based matcher	0.136	-0.043	0.112	**-0.038**	**0.263**	-0.236
Abstract Matcher	0.047	0.182	0.134	**0.286**	**0.205**	0.152
mean	0.022	0.158	-0.021	0.120	**0.330**	**0.229**

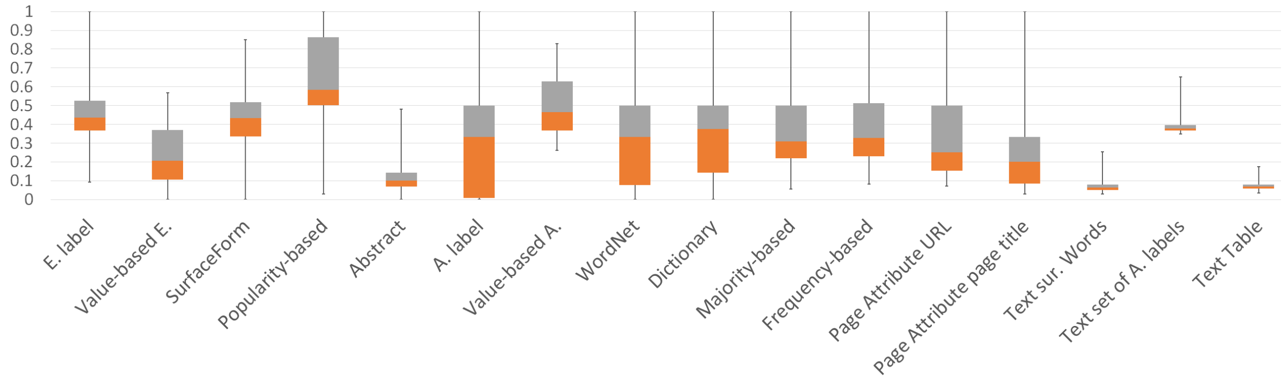


Figure 5: Matrix aggregation weights

similarity scores, we use a 1 : 1 decisive second line matcher for generating correspondences. The matcher selects for each entity/attribute/table the candidate with the highest similarity score. This score needs to be above a certain threshold in order to ensure that correspondences are only generated if the matching system is certain enough. The thresholds are determined for each combination of matchers using decision trees and 10-fold-cross-validation. In addition to thresholding, we also apply the filtering rule that we only generate correspondences for a table if (1) a minimum of three entities in the table have a correspondence to an instance in the knowledge base and (2) one forth of the entities in the table is matched to instances of the class we decided for.

We evaluate the matching results according to precision, recall and F1. We compare our results to the results of existing approaches. However, it is tricky to directly compare result as the other systems were tested using different web tables and different knowledge bases and as the difficulty of the matching task is highly dependent on these inputs.

8.1 Row-to-Instance Matching Results

Table 4 presents the results of the row-to-instance matching task for different combinations of matchers. If we only considering the entity label feature, a moderate result with a precision of 0.72 is achieved. Also taking the table cell values into account increases the recall by 0.09 and the precision by 0.08. As expected, based on the weight analysis, considering the values helps to improve the performance but only using the entity label already leads to a decent amount of correct correspondences. By adding surface forms, the recall can again be raised by 0.02 which indicates that we indeed find alternative names for entities in the tables. The popularity-based matcher can slightly increase the precision and recall. Whenever the similarities for candidate instances are close, to decide for the more common one is in most cases the better decision. However, this assumption does especially not hold for web tables containing long-tail entities, e.g. entities that are rather unknown.

Including the only instance matcher relying on a multiple table feature (Abstract matcher), the precision is strongly increased by 0.13 while 0.08 recall is lost. This might be unexpected at first glance since a matcher comparing bag-of-words tends to add a lot of noise. The reason is the choice of the threshold since it needs to be very high to prevent a breakdown of the F1 score. Thus, comparing the entity as a whole with the DBpedia abstracts helps to find correct correspondences but has to be treated with caution to not ruin the overall performance. If we use the combination of all instance matchers, the highest F1 value can be achieved. This shows that the instances found by matchers exploiting different features do not necessarily overlap and that the matchers can benefit from each other by compensating their weaknesses.

In the following, we compare our results to existing results

Table 4: Row-to-instance matching results

Matcher	P	R	F1
Entity label matcher	0.72	0.65	0.68
Entity label matcher + Value-based entity matcher	0.80	0.74	0.77
Surface form matcher + Value-based entity matcher	0.80	0.76	0.78
Entity label matcher + Value-based entity matcher + Popularity-based matcher	0.81	0.76	0.79
Entity label matcher + Value-based entity matcher + Abstract matcher	0.93	0.68	0.79
All	0.92	0.71	0.80

from literature. While Mulwad et al. [26] report an accuracy of 0.66 for a pure label-based instance matching approach, the F1 score achieved by Limaye et al. [22] (web manual data set) is 0.81 when taking alternative names for the labels into account. Extending the label-based method by including the values results in an accuracy of 0.77 [38] resp. a F1 score of 0.82 if the web tables are matched to DBpedia and 0.89 if they are matched against Yago [25]. Very high F1 scores above 0.9 are stated by Zhang [42]. However, the presented baseline that only queries the Freebase API already obtains very close scores such that the good performance is mainly due to the internal API ranking. For other APIs used by the systems, it is not always clear which ranking functions are used and which performance they already achieve without considering any other features.

8.2 Attribute-To-Property Matching Results

Table 5 shows the results of our attribute-to-property matching experiments using different combinations of matchers. In contrast to the row-to-instance matching task, we get a rather low recall (0.49) if we only take the attribute label into account. Based on the weight analysis, we already know that the attribute label is not necessarily a useful feature for all the tables. Including cell values increases the recall by 0.35 but decreases the precision by 0.10. While it provides the possibility to compensate non-similar labels, it also adds incorrect correspondences if values accidentally fit. This especially holds for attributes of data type numeric and date, for example, in a table describing medieval kings it will be quite difficult to distinguish birth dates and death dates by only examining a single attribute at a time. Nevertheless, the values present a valuable feature especially to achieve a decent level of recall, given that the attribute labels are often misleading. Taking WordNet into account does neither improve precision nor recall. This shows, that a general dictionary is not very useful for the property matching task. In contrast, using the dictionary created from web tables increases the recall as well as the precision. With specific background knowledge that is tailored to the web tables, it is possible to enhance the performance. However, the creation of the dictionary requires a lot of smart filtering. Without proper filtering, the dictionary would add only noise. The result of using all matchers together is slightly lower than the best result due to the WordNet matcher.

Our results for the attribute-to-property matching task are difficult to compare to other existing results as many of the existing systems only match attributes to object properties and do not cover data type properties, such as numbers and dates. For this task, Mulwad et al. [26] report an accuracy of 0.25, their advanced system achieves a F1 score of 0.89 [25] while Muñoz et al. [24] report a F1 score of 0.79. Although Limaye et al. [22] additionally include the attribute header, only a result of 0.52 (F1) can be reached. Even without the consideration of data type properties, the property matching task seems to be more difficult than the instance matching task.

8.3 Table-To-Class Matching Results

Table 6 reports the results of our table-to-class matching experiments. Since we need an instance similarity matrix for the class matching, we use the entity label matcher together with the valued-based matcher in all following experiments. When only considering the majority of the instance correspondences to compute the class correspondences, the precision is 0.47 and the recall 0.51, meaning that only for approximately half of the tables the correct class is assigned. One reason for this is the preferential treatment of superclasses over specific classes which are further down in the class hierarchy. All instances that can be found in a specific class are also contained in the superclass and there might be further instances belonging to the superclass that fit. Together with the consideration of the frequency which exactly tackles the mentioned issue, a F1 score of 0.89 can be reached.

In order to see how far we get when solely considering matchers that rely on context features, we evaluate the page attribute matcher and the text matcher independently from the others. Since the differences in the performance are marginal, we do not present the results for the individual features. Whenever the page attribute matcher finds a correspondence, this correspondence is very likely to be correct. However, since the URL and page title are compared with the label of the class, it can happen that no candidate is found at all. Regarding the recall, similar holds for the text matcher but the generated correspondences are not necessarily correct. This is not surprising because we already discovered that matchers using features represented as bag-of-words have a weak ability to differentiate between correct and incorrect candidates due to a lot of noise.

When we combine all previous matchers, a F1 of 0.88 is obtained which is still lower than the outcome of the majority-based together with the frequency-based matcher. If we make use of the number of available class matchers which is transposed by the agreement matcher, we reach a F1 value of 0.92. Thus, taking advantage of features covering the whole spectrum of available information and deciding for the class most of them agree on, is the best strategy for the class matching task.

Due to the fact that the table-to-class matching task has a strong influence on the other two matching tasks in $T2KMatch$, their performance can be substantially reduced

Table 5: Attribute-to-property matching results

Matcher	P	R	F1
Attribute label matcher	0.85	0.49	0.63
Attribute label matcher + Duplicate-based attribute matcher	0.75	0.84	0.79
WordNet matcher + Duplicate-based attribute matcher	0.71	0.83	0.77
Dictionary matcher + Duplicate-based attribute matcher	0.76	0.86	0.81
All	0.70	0.84	0.77

Table 6: Table-to-class matching results

Matcher	P	R	F1
Majority-based matcher	0.47	0.51	0.49
Majority-based matcher + Frequency-based matcher	0.87	0.90	0.89
Page attribute matcher	0.97	0.37	0.53
Text matcher	0.75	0.34	0.46
Page attribute matcher + Text matcher +			
Majority-based matcher + Frequency-based matcher	0.9	0.86	0.88
All	0.93	0.91	0.92

whenever a wrong class decision is taken. For example, when solely using the text matcher, the row-to-instance recall drops down to 0.52 and the attribute-to-property recall to 0.36.

For the table-to-class matching task, results between 0.43 (F1) [22] and 0.9 (accuracy) [38] are reported in the literature. In between, we find outcomes varying from 0.55 (F1) [43] over 0.65 to 0.7 for different knowledge bases [39]. When taking also the specificity of the classes into account, the performance of 0.57 (F1) is neither higher nor lower than other results [25]. Similar holds for considering the context with a result of 0.63 (F1) [42].

In summary, our matching system is able to distinguish between tables that can be matched to DBpedia and tables that do not have any counterparts. This becomes especially obvious if we look at the results of the table-to-class matching task. The ability to properly recognize which tables can be matched is a very important characteristic when dealing with web tables. Whenever the table can be matched to DBpedia, features directly found in the table are crucial for the instance and property matching tasks. For properties, the cell values need to be exploited in order to achieve an acceptable recall. Adding external resources is useful the closer the content of the external resource is related to the web tables or to the knowledge base. For the table-to-class matching task, the majority of instances as well as the specificity of a class has a very high impact on the performance. While matchers based on page attributes often do not find a correspondence at all, this is the opposite for all features represented as bag-of-words which add a large amount of noise. Ways to handle the noise are either filtering or to only use them as an additional indicator whenever matchers based on other features agree with the decision.

Comparing the related work among each other shows that almost no conclusions can be drawn whether a certain feature is useful for a matching task or not. One reason for this is that the systems are applied to different sets of web tables and different knowledge bases. As indicated by Hassanzadeh et al. [16], the choice of the knowledge base has a strong influence on the matching results. For example, a knowledge base might not contain certain instances (e.g. web tables contain a lot of product data while DBpedia hardly covers products) or properties at all (e.g. product prices) and the granularity of the classes can differ a lot, depending on the structure and the focus of the knowledge base [16, 31].

9. CONCLUSION

This paper studied the utility of different features for task of matching web tables against a knowledge base. We provided an overview as well as a classification of the features used in state-of-the-art systems. The features can either be found in the table itself or in the context of the table. For each of the features, we introduce task specific matchers that compute similarities to instances, properties, and classes in a knowledge base. The resulting similarity matrices, representing the feature-specific results, have been combined using matrix predictors in order to gain insights about the suitability of the aggregation weights. Using matrix predictors, we allow different web tables to favor the features that are most suitable for them.

We showed that a positive correlation between the weighting based on the reliability scores as well as the performance measures precision and recall can be found. However, the best way to compute reliability scores differs depending on the matching task. While predictors based on the diversity of the matrix elements work best for the row-to-instance and table-to-class matching task, an average-based predictor shows a better performance for the attribute-to-property matching task.

The computed weights gave us an idea which features are in general important for the individual matching tasks and how much their significance varies between the tables. While the entity label and the popularity of an instance are very important for the row-to-instance matching task, comparing the cell values is crucial for the attribute-to-property matching task. For the table-to-class matching task, several features are important, while the ones directly coming from the table outperform context features. The largest variation in the weights was discovered for the attribute labels. This indicates that attribute labels can be a good feature as long as meaningful attribute names are found in the web tables but also that this is not always the case.

We further explored the performance of different ensembles of matchers for all three matching tasks. In summary, taking as many features as possible into account is promising for all three tasks. Features found within tables generally lead to the best results than context features. Nevertheless, taking context features into account can improve the results but particular caution is necessary since context features may also add a lot of noise. External resources proofed to useful as long as their content is closely related to the content of the web tables, i.e. the general lexical database WordNet did not improve the results for the attribute-to-property matching task while a more specific dictionary did improve the results. The performance that we achieved in our experiments for the row-to-instance and the attribute-to-property matching tasks are roughly in the same range as the results reported in literature. For the table-to-class matching task, our results are higher than the ones reported in the related work.

The source code of the extended version of the *T2KMatch* matching framework that was used for the experiments is found on the *T2K* website[4]. The gold standard that was used for the experiments can be downloaded from the Web Data Commons website[5].

10. REFERENCES

[1] K. Braunschweig, M. Thiele, J. Eberius, and W. Lehner. Column-specific Context Extraction for Web Tables. In *Proc. of the 30th Annual ACM Symposium on Applied Computing*, pages 1072–1077, 2015.

[2] V. Bryl, C. Bizer, and H. Paulheim. Gathering alternative surface forms for dbpedia entities. In *Proceedings of the Third NLP&DBpedia Workshop (NLP & DBpedia 2015)*, pages 13–24, 2015.

[3] M. J. Cafarella, A. Halevy, and N. Khoussainova. Data Integration for the Relational Web. *Proc. of the VLDB Endow.*, 2:1090–1101, 2009.

[4] M. J. Cafarella, A. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. WebTables: Exploring the Power of Tables on the Web. *Proc. of the VLDB Endow.*, 1:538–549, 2008.

[5] I. F. Cruz, F. P. Antonelli, and C. Stroe. Efficient selection of mappings and automatic quality-driven combination of matching methods. In *Proc. of the 4th Int. Workshop on Ontology Matching*, 2009.

[6] I. F. Cruz, V. R. Ganesh, and S. I. Mirrezaei. Semantic Extraction of Geographic Data from Web Tables for Big Data Integration. In *Proc. of the 7th Workshop on Geographic Information Retrieval*, pages 19–26, 2013.

[7] J. Daiber, M. Jakob, C. Hokamp, and P. N. Mendes. Improving efficiency and accuracy in multilingual entity extraction. In *Proc. of the 9th Int. Conference on Semantic Systems*, 2013.

[8] A. Das Sarma, L. Fang, N. Gupta, A. Halevy, H. Lee, F. Wu, R. Xin, and C. Yu. Finding Related Tables. In *Proc. of the Int. Conference on Management of Data*, 2012.

[9] H.-H. Do and E. Rahm. COMA: A System for Flexible Combination of Schema Matching Approaches. In *Proc. of the 28th Int. Conference on Very Large Data Bases*, pages 610–621, 2002.

[10] 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 *Proc. of the 20th SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 601–610, 2014.

[11] J. Euzenat and P. Shvaiko. *Ontology Matching.* Springer, 2007.

[12] C. Fellbaum. *WordNet: An Electronic Lexical Database.* MIT Press, 1998.

[13] A. Gal, H. Roitman, and T. Sagi. From Diversity-based Prediction to Better Ontology & Schema Matching. In *Proc. of the 25th Int. World Wide Web Conference*, 2016.

[14] A. Gal. *Uncertain Schema Matching.* Synthesis Lectures on Data Management. Morgan & Claypool, 2011.

[15] A. Gal and T. Sagi. Tuning the ensemble selection process of schema matchers. *Information Systems*, 35(8):845 – 859, 2010.

[16] O. Hassanzadeh, M. J. Ward, M. Rodriguez-Muro, and K. Srinivas. Understanding a large corpus of web tables through matching with knowledge bases: an empirical study. In *Proc. of the 10th Int. Workshop on Ontology Matching*, 2015.

[17] 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.

[18] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia. *Semantic Web Journal*, 6(2):167–195, 2015.

[19] O. Lehmberg, D. Ritze, R. Meusel, and C. Bizer. A Large Public Corpus of Web Tables containing Time and Context Metadata. In *Proc. of the 25th International World Wide Web Conference*, 2016.

[20] O. Lehmberg and C. Bizer. Web table column categorisation and profiling. In *Proc. of the 19th International Workshop on Web and Databases*, pages 4:1–4:7, 2016.

[21] O. Lehmberg, D. Ritze, P. Ristoski, R. Meusel, H. Paulheim, and C. Bizer. The Mannheim Search Join Engine. *Web Semantics: Science, Services and Agents on the World Wide Web*, 35:159–166, 2015.

[22] G. Limaye, S. Sarawagi, and S. Chakrabarti. Annotating and Searching Web Tables Using Entities, Types and Relationships. *Proc.of the VLDB Endow.*, 3:1338–1347, 2010.

[23] X. Ling, A. Halevy, F. Wu, and C. Yu. Synthesizing union tables from the web. In *Proc. of the 23rd Int. Joint Conference on Artificial Intelligence*, pages 2677–2683, 2013.

[24] E. Muñoz, A. Hogan, and A. Mileo. Using Linked Data

[4] http://dws.informatik.uni-mannheim.de/en/research/T2K

[5] http://webdatacommons.org/webtables/goldstandardV2.html

to Mine RDF from Wikipedia's Tables. In *Proc. of the 7th ACM Int. Conference on Web Search and Data Mining*, pages 533–542, 2014.

[25] V. Mulwad, T. Finin, and A. Joshi. Semantic message passing for generating linked data from tables. In *Proc. of the 12th Int. Semantic Web Conference*, 2013.

[26] V. Mulwad, T. Finin, Z. Syed, and A. Joshi. Using linked data to interpret tables. In *Proc. of the 1st Int. Workshop on Consuming Linked Data*, 2010.

[27] S. Neumaier, J. Umbrich, J. X. Parreira, and A. Polleres. Multi-level semantic labelling of numerical values. In *Proc. of the 15th International Semantic Web Conference*, pages 428–445, 2016.

[28] K. Pearson. Notes on regression and inheritance in the case of two parents. *Proc. of the Royal Society of London*, 58:240–242, 1895.

[29] S. Rhoades. The Herfindahl-Herschman Index. *Federal Reserve Bulletin*, 79:188–189, 1993.

[30] D. Rinser, D. Lange, and F. Naumann. Cross-Lingual Entity Matching and Infobox Alignment in Wikipedia. *Information Systems*, 38:887–907, 2013.

[31] D. Ritze, O. Lehmberg, Y. Oulabi, and C. Bizer. Profiling the Potential of Web Tables for Augmenting Cross-domain Knowledge Bases. In *Proc. of the 25th International World Wide Web Conference*, 2016.

[32] D. Ritze, O. Lehmberg, and C. Bizer. Matching HTML Tables to DBpedia. In *Proc. of the 5th International Conference on Web Intelligence, Mining and Semantics*, 2015.

[33] T. Sagi and A. Gal. Schema matching prediction with applications to data source discovery and dynamic ensembling. *VLDB Journal*, 22:689–710, 2013.

[34] G. Salton and M. McGill. *Introduction to modern information retrieval.* McGraw-Hill, 1983.

[35] Y. A. Sekhavat, F. di Paolo, D. Barbosa, and P. Merialdo. Knowledge Base Augmentation using Tabular Data. In *Proc. of the 7th Workshop on Linked Data on the Web*, 2014.

[36] A. Singhal. Introducing the knowledge graph: Things, not string. Blog, 2012. Retrieved March 19, 2015.

[37] F. Suchanek, S. Abiteboul, and P. Senellart. Paris: Probabilistic alignment of Relations, Instances, and Schema. *Proc. VLDB Endowment*, 5:157–168, 2011.

[38] Z. Syed, T. Finin, V. Mulwad, and A. Joshi. Exploiting a Web of Semantic Data for Interpreting Tables. In *Proc. of the 2nd Web Science Conference*, 2010.

[39] P. Venetis, A. Halevy, J. Madhavan, M. Paşca, W. Shen, F. Wu, G. Miao, and C. Wu. Recovering Semantics of Tables on the Web. *Proc. of the VLDB Endow.*, 4(9):528–538, 2011.

[40] J. Wang, H. Wang, Z. Wang, and K. Q. Zhu. Understanding Tables on the Web. In *Proc. of the 31st Int. Conf. on Conceptual Modeling*, pages 141–155, 2012.

[41] M. Yakout, K. Ganjam, K. Chakrabarti, and S. Chaudhuri. InfoGather: Entity Augmentation and Attribute Discovery by Holistic Matching with Web Tables. In *Proc. of the 2012 SIGMOD*, pages 97–108, 2012.

[42] Z. Zhang. Towards efficient and effective semantic table interpretation. In *Proc. of the 13th International Semantic Web Conference*, pages 487–502. 2014.

[43] S. Zwicklbauer, C. Einsiedler, M. Granitzer, and C. Seifert. Towards disambiguating Web tables. In *Proc. of the 12th Int. Semantic Web Conference*, pages 205–208, 2013.

Schema Inference for Massive JSON Datasets

Mohamed-Amine Baazizi
LIP6
Université Pierre et Marie Curie
Mohamed-Amine.Baazizi@lip6.fr

Houssem Ben Lahmar
IPVS
University of Stuttgart
Houssem.Ben-Lahmar@ipvs.uni-stuttgart.de

Dario Colazzo
Université Paris-Dauphine,
PSL Research University,
CNRS, LAMSADE
75016 PARIS, FRANCE
dario.colazzo@dauphine.fr

Giorgio Ghelli
Dipartimento di Informatica
Università di Pisa
ghelli@di.unipi.it

Carlo Sartiani
DIMIE
Università della Basilicata
sartiani@gmail.com

ABSTRACT

In the recent years JSON affirmed as a very popular data format for representing massive data collections. JSON data collections are usually schemaless. While this ensures several advantages, the absence of schema information has important negative consequences: the correctness of complex queries and programs cannot be statically checked, users cannot rely on schema information to quickly figure out the structural properties that could speed up the formulation of correct queries, and many schema-based optimizations are not possible.

In this paper we deal with the problem of inferring a schema from massive JSON datasets. We first identify a JSON type language which is simple and, at the same time, expressive enough to capture irregularities and to give complete structural information about input data. We then present our main contribution, which is the design of a schema inference algorithm, its theoretical study, and its implementation based on Spark, enabling reasonable schema inference time for massive collections. Finally, we report about an experimental analysis showing the effectiveness of our approach in terms of execution time, precision, and conciseness of inferred schemas, and scalability.

CCS Concepts

•**Information systems** → **Semi-structured data; Data model extensions;** •**Theory of computation** → *Type theory; Logic;*

Keywords

JSON, schema inference, map-reduce, Spark, big data collections

1. INTRODUCTION

Big Data applications typically process and analyze very large structured and semi-structured datasets. In many of these applications, and in those relying on NoSQL document stores in particular, data are represented in JSON (JavaScript Object Notation) [10], a data format that is widely used thanks to its flexibility and simplicity.

JSON data collections are usually schemaless. This ensures several advantages: in particular it enables applications to quickly consume huge amounts of semi-structured data without waiting for a schema to be specified. Unfortunately, the lack of a schema makes it impossible to statically detect unexpected or unwanted behaviours of complex queries and programs (i.e., lack of correctness), users cannot rely on schema information to quickly figure out structural properties that could speed up the formulation of correct queries, and many schema-based optimizations are not possible.

In this paper we deal with the problem of inferring a schema from massive JSON datasets. Our main goal in this work is to infer structural properties of JSON data, that is, a description of the structure of JSON objects and arrays that takes into account nested values and the presence of optional values. These are the main properties that characterize semi-structured data, and having a tool that ensures *fast*, *precise*, and *concise* inference is crucial in modern applications characterized by agile consumption of huge amounts of data coming from multiple and disparate sources.

The approach we propose here is based on a JSON schema language able to capture structural irregularities and complete structural information about input data. This language resembles and borrows mechanisms from existing proposals [20], but it has the advantage to be simple yet very expressive.

The proposed technique infers a schema that provides a *global* description of the whole input JSON dataset, while having a size that is small enough to enable a user to consult it in a reasonable amount of time, in order to get a global knowledge of the structural and type properties of the JSON collection. The description of the input JSON collection is global in the sense that each path that can be traversed in the tree-structure of each input JSON value can be traversed in the inferred schema as well. This property is crucial to enable a series of query optimization tasks. For instance,

Series ISSN: 2367-2005

10.5441/002/edbt.2017.21

thanks to this property JSON queries [1, 9] can be optimized at compile-time by means of schema-based path rewriting and wildcard expansion [16] or projection [8]. These optimizations are not possible if the schema hides some of the structural properties of the data, as happens in related approaches [22].

At the same time, our inferred schemas precisely capture the presence of optional and mandatory fields in collection of JSON records. Thanks to our approach, the user has a precise knowledge about i) all possible fields of records, ii) optional ones, and iii) mandatory ones. Property i) is crucial, as thanks to it the user can avoid time consuming, error-prone (approximated) data explorations to realize what fields can be really selected, while property ii) guides the user towards the adoption of code to handle the optional presence of certain fields; property iii), finally, indicates fields that can be always selected for each record in the collection.

A precise schema, like the one that can be inferred by our approach, can be very useful when very large datasets must be analyzed or queried with main-memory tools: indeed, by identifying the data requirements of a query or a program through a simple static analysis technique, it is possible to match these requirements with the schema in order to load in main memory only those fragments of the input dataset that are actually needed, hence improving both scalability and performance.

It is worth stressing that, even if in some cases JSON data feature a rather regular structure, the only alternative way for the user to be sure that all possible (optional) fields are identified is to explore the entire dataset either manually or by means of scripts that must be manually adapted to each particular JSON source, with weak guarantees of efficiency and soundness. Our approach instead applies to any JSON data collection, and is shown to be sound and effective on massive datasets. In addition, it is worth observing that, while in many cases processed JSON data come from remote, uncontrolled sources, in other particular cases JSON data are generated by applications whose code is known. In these cases a wider knowledge is available about the structure of the program output, but again schema inference is important as it can highlight subtle structural properties that can arise only in outputs of some particular program runs; also, when the code starts being complex, it is difficult to precisely figure out the structure of output JSON data. In some other cases, remote JSON sources can be accessed by APIs (e.g., Twitter APIs) that sometimes are provided with some schema descriptions. Unfortunately, these descriptions are often incomplete, some fields are often ignored, and the distinction between optional and mandatory fields is often omitted.

Our Contribution. Our main contribution is the design of a schema inference algorithm and its implementation based on Apache Spark [7], in order to ensure reasonable schema inference time for massive collections. Our schema inference approach consists of two main steps. In the first one, an input collection of JSON values is processed by a Map transformation in order to infer a simple type for each value. The resulting output is processed by a Reduce action, which *fuses* inferred types that are not necessarily identical, but that share similar structure. This step relies on a binary function that takes two JSON types as input and fuses them. This function inspects the two input types and identifies parts that are mandatory, optional, or repeated in the types, in order to obtain a type which is a super type of the two input types (it includes them), but that is potentially much more succinct than their simple union. A theoretical study shows that the fusion function is correct and, very importantly, associative.

Associativity is crucial as it allows Spark to safely distribute and parallelize the fusion of a massive collection of values. Associativity is also important to enable incremental evolution of the inferred schema under updates. In many applications the JSON sources are dynamic, and new values can be added at any time, with a structure that can differ from that already inferred for previous records. In this situation, in the case of insertion of a new record in an existing record collection, thanks to associativity, we simply need to fuse the existing schema with the schema of the new record. For incremental maintenance under other forms of updates, in the usual case that a massive dataset is kept partitioned and the updated parts are known, it just suffices to re-infer the schema for the updated parts and to fuse them with previously inferred schemas for unchanged parts.

Our last contribution consists of an implementation of the proposed approach based on Spark, as well as an experimental evaluation validating our claims of succinctness, precision, and efficiency. We based our tests on 4 real JSON datasets. Our experiments confirm that our schema inference algorithm returns very succinct yet precise schemas, even in the presence of poorly organized data (i.e., Wikipedia dataset). Furthermore, a scalability analysis reveals that our approach ensures reasonable execution times, and that a simple partitioning strategy allows the performance to be improved.

Paper Outline. The paper is organized as follows. In Section 2 we illustrate some scenarios that motivate our work. In Section 3, then, we survey existing works. In Section 4, we describe the data model and the schema language we use here, while in Section 5 we present our schema inference approach. In Sections 6 and 7, finally, we show the results of our experimental evaluation and draw our conclusions.

2. MOTIVATION AND OVERVIEW

This section overviews the two steps of our schema fusion approach: type inference and type fusion. To this end, we first briefly recall the general syntax and semantics of JSON values. As in most semi-structured models, JSON distinguishes between basic values, which range over numbers (e.g., 123), strings (e.g., "abc"), and booleans (i.e., true/-false), and complex values which can be either (unordered) sets of key/value pairs called *records* or (ordered) lists of values called *arrays*. The only constraint that JSON values must obey is key uniqueness within each record. Arrays can mix both basic and complex types. In the following, we will use the term *mixed-content arrays* for arrays mixing atomic and complex values.

A sample JSON record is illustrated in Figure 1. Syntactically, records use the conventional curly braces symbols whereas arrays use square brackets; finally, string values and keys are wrapped inside quotes in JSON (but we will avoid quotes around keys in our formal syntax).

```
{"A": 123
 "B": "The ..."
 "C": false
 "D": ["abc", "cde", "fr12"]
}
```

Figure 1: A JSON record r_1.

Type inference.

Type inference, during the Map phase, is dedicated to inferring individual types for the input JSON values, and yields a set of distinct types to be fused during the Reduce phase. Some proposals of JSON schemas exist in the literature. With one exception [20], none of them uses regular expressions which, as we shall illustrate, are important for concisely representing types for array values. Moreover, a clean formal semantics specification of types is often missing in these works, hence making it difficult to understand their precise meaning.

The type language we adopt is meant to capture the core features of the JSON data model with an emphasis on succinctness. Intuitively, basic values are captured using standard data types (i.e., String, Number, Boolean), complex values are captured by introducing record and array type constructors, and a union type constructor is used to add flexibility and expressive power. To illustrate the type language, observe the following type that is inferred for the record r_1 given in Figure 1:

$$\{A : \texttt{Num}, B : \texttt{Str}, C : \texttt{Bool}, D : [\texttt{Str}, \texttt{Str}, \texttt{Str}]\}$$

As we will show, the initial type inference is a quite simple and fast operation: it consists of a simple traversal of the input values that produces a type that is isomorphic to the value itself.

Type fusion.

Type fusion is the second step of our approach and consists in iteratively merging the types produced during the Map phase. Because it is performed during the Reduce phase in a distributed fashion, type fusion relies on a fusion operator which enjoys the commutativity and associativity properties. This fusion operator is invoked over two types T_1 and T_2, and produces a supertype of the inputs. To do so, the fusion collapses the parts of T_1 and T_2 that are identical and preserves the parts that are distinct in both types. To this end, T_1 and T_2 are processed in a synchronised top-down manner in order to identify common parts. The main idea is to represent only once what is common, and, at the same time, to preserve all the parts that differ.

Fusion treats atomic types, record types, and array types differently, as follows.

- Atomic types: the fusion of atomic types is obvious, as identical types are collapsed while different types are combined using the union operator.

- Record types: recall that valid record types enjoy key uniqueness. Therefore, the fusion of T_1 and T_2 is led by two rules:

  (R_1) matching keys from both types are collapsed and their respective types are recursively fused;

  (R_2) keys without a match are deemed optional in the resulting type and decorated with a question mark ?.

  To illustrate those cases, assume that T_1 and T_2 are, respectively, $\{A{:}\texttt{Str}, B{:}\texttt{Num}\}$ and $\{B{:}\texttt{Bool}, C{:}\texttt{Str}\}$. The only matching key is "B" and hence its two atomic types Num and Bool are fused, which yields $\texttt{Num}+\texttt{Bool}$. The other keys will be optional according to rule R_2. Hence, fusion yields the type

  $$T_{12} = \{(A{:}\texttt{Str})?, B{:}\texttt{Num} + \texttt{Bool}, (C{:}\texttt{Str})?\}$$

  Assume now that T_{12} is fused with

  $$T_3 = \{A{:}\texttt{Null}, B{:}\texttt{Num}\}$$

  Rules R_1 and R_2 need to be slightly adapted to deal with optional types. Intuitively, we should simply consider that optionality '?' prevails over the implicit *total* cardinality '1'. The resulting type is thus

  $$T_{123} = \{(A{:}\texttt{Str} + \texttt{Null})?, B{:}\texttt{Num} + \texttt{Bool}, (C{:}\texttt{Str})?\}.$$

  Fusion of nested records eventually associates keys with types that may be unions of atomic types, record types, and array types. We will see that, when such types are merged, we separately merge the atomic types, the record types, and the array types, and return the union of the result. For instance, the fusion of types

  $$\{l{:}(\texttt{Bool} + \texttt{Str} + \{A{:}\texttt{Num}\}\}$$

  and

  $$\{l{:}(\texttt{Bool} + \{A{:}\texttt{Str}, B{:}\texttt{Num}\})\}$$

  yields

  $$\{l{:}(\texttt{Bool} + \texttt{Str} + \{A{:}(\texttt{Num} + \texttt{Str}), (B{:}\texttt{Num})?\}\}.$$

- Array types: array fusion deserves special attention. A particular aspect to consider is that an array type obtained in the first phase may contain several repeated types, and may feature mixed-content. To deal with this, before fusing types we perform a kind of simplification on bodies by using regular expression types, and, in particular, union + and repetition $*$. To illustrate this point, consider the array value

  $$[''abc'','' cde'', \{''E'' :''fr'', \ ''F'' : 12\}],$$

  containing two strings followed by a record (mixed-content). The first phase infers for this value the type

  $$[\texttt{Str}, \texttt{Str}, \{E{:}\texttt{Str}, F{:}\texttt{Num}\}].$$

  This type can be actually simplified. For instance, one can think of a *partition*-based approach which collapses adjacent identical types into a star-guarded type, thus transforming

  $$[\texttt{Str}, \texttt{Str}, \{E{:}\texttt{Str}, F{:}\texttt{Num}\}]$$

  into

  $$[(\texttt{Str})*, \{E{:}\texttt{Str}, F{:}\texttt{Num}\}]$$

  by collapsing the string types. The resulting schema is indeed succinct and precise. However, succinctness cannot be guaranteed after fusion. For instance, if that type was to be merged with

  $$[\{E{:}\texttt{Str}, F{:}\texttt{Num}\}, \texttt{Str}, \texttt{Str}],$$

where strings and record swapped positions, succinctness would be lost because we need to duplicate at least one sub-expression, $(\texttt{Str})*$ or $\{E{:}\texttt{Str}, F{:}\texttt{Num}\}$. As we are mainly concerned with generating types that can be human-readable, we trade some precision for succinctness and do not account for position anymore. To achieve this, in our simplification process (made before fusing array types) we generalize the above partition-based solution by returning the star-guarded union of all distinct types expressed in an array. So, simplification for either

$$[\texttt{Str}, \texttt{Str}, \{E{:}\texttt{Str}, F{:}\texttt{Num}\}]$$

or

$$[\{E{:}\texttt{Str}, F{:}\texttt{Num}\}, \texttt{Str}, \texttt{Str}]$$

yields the same type

$$S = [(\texttt{Str} + \{E{:}\texttt{Str}, F{:}\texttt{Num}\})*].$$

After the array types have been simplified in this manner, they are fused by simply recursively fusing their content types, applying the same technique described for record types: when the body type is a union type, we separately merge the atomic components, the array components, and the record components, and take the union of the results.

3. RELATED WORK

The problem of inferring structural information from JSON data collections has recently gained attention of the database research community. The closest work to ours is the very preliminary investigation that we presented in [12]. While [12] only provides a sketch of a MapReduce approach for schema inference, in this paper we present results about a much deeper study. In particular, while in [12] a declarative specification of only a few cases of the fusion process is presented, in this paper we fully detail this process, provide a formal specification as well as a fusion algorithm. Furthermore, differently from [12], we present here an experimental evaluation of our approach validating our claims of parallelizability and succinctness.

In [22] Wang et al. present a framework for efficiently managing a schema repository for JSON document stores. The proposed approach relies on a notion of JSON schema called *skeleton*. In a nutshell, a skeleton is a collection of trees describing structures that frequently appear in the objects of JSON data collection. In particular, the skeleton may totally miss information about paths that can be traversed in some of the JSON objects. In contrast, our approach enables the creation of a complete yet succinct schema description of the input JSON dataset. As already said, having such a complete structural description is of vital importance for many tasks, like query optimisation, defining and enforcing access-control security policies, and, importantly, giving the user a global structural vision of the database that can help her in querying and exploring the data in an effective way. Another important application of complete schema information is query type checking: as illustrated in [12] our inferred schemas can be used to make type checking of Pig Latin scripts much stronger.

In a very recent work [20], motivated by the need of laying the formal foundations for the JSON Schema language [3], Pezoa et al. present the formal semantics of that language, as well as a theoretical study of its related expressive power and validation problem. While that work does not deal with the schema inference problem, our schema language can be seen as a core part of the JSON Schema language studied therein, and shares union types and repetition types with that one. These constructors are at the basis of our technique to collapse several schemas into a more succinct one. An alternative proposal for typing JSON data is JSound [2]. That language is quite restrictive wrt ours and JSON Schemas: for instance it lacks union types.

In a very recent work [13] Abadi and Discala deal with the problem of automatic transforming denormalised, nested JSON data into normalised relational data that can be stored into a RDBMS; this is achieved by means of a schema generation algorithm that learns the *normalised, relational* schema from data. Differently from that work, we deal with schemas that are far from being relational, and are closer to tree regular grammars [17]. Furthermore, the approach proposed in [13] ignores the original structure of the JSON input dataset and, instead, depends on patterns in the attribute data values (functional dependencies) to guide its schema generation. So, that approach is complementary to ours.

In [15] Liu et al. propose storage, querying, and indexing principles enabling RDBMSs to manage JSON. The paper does not deal with schema inference, but indicates a possible optimisation of their framework based on the identification of common attributes in JSON objects that can be captured by a relational schema for optimization purposes. In [21] Scherzinger et al. propose a plugin to track changes in object-NoSQL mappings. The technique is currently limited to only detect mismatches between base types (e.g., Boolean, Integer, String), and the authors claim that a wider knowledge of schema information is needed to enable the detection of other kinds of changes, like, for instance, the removal or renaming of attributes.

It is important to state that the problem of schema inference has already been addressed in the past in the context of semi-structured and XML data models. In [18] and [19], Nestorov et al. describe an approach to extract a schema from semistructured data. They propose an object-oriented type system where nodes are captured by classes built starting from nodes sharing the same incoming and outcoming edges and where data edges are generalized to relations between the classes. In [19], the problem of building a type out a of a collection of semistructured documents is studied. The emphasis is put on minimizing the size of the resulting type while maximizing its precision. Although that work considers a very general data model captured by graphs, it does not suit our context. Firstly, we consider the JSON model, that is tree-shaped by nature and that features specific constructs such as arrays that are not captured by the semi-structured data model. Secondly, we aim at processing potentially large datasets efficiently, a problem that is not directly addressed in [18] and [19].

More recent efforts on XML schema inference (see [14] and works cited therein) are also worth mentioning since they are somewhat related to our approach. The aim of these approaches is to infer restricted, yet expressive enough forms of regular expressions starting from a positive set of strings representing element contexts of XML documents. While XML and JSON both allow one to represent tree-shaped data, they have radical differences that make existing XML related approaches difficult to apply to the JSON setting.

Similar remarks hold for related approaches for schema inference for RDF [11]. Furhermore, none of these approaches is designed to deal with massive datasets.

4. DATA MODEL AND TYPE LANGUAGE

This section is devoted to formalizing the JSON data model and the schema language we adopt.

We represent JSON values as records and arrays, whose abstract syntax is given in Figure 2. Basic values B comprise *null* value, booleans, numbers n, and strings s. As outlined in Section 2, records are sets of fields, each field being an association of a value V to a key l whereas arrays are sequences of values. The abstract syntax is practical for the formal treatment, but we will typically use the more readable notation introduced at the bottom of Figure 2, where records as represented as $\{l_1 : V_1, \ldots, l_n : V_n\}$ and arrays are represented as $[V_1, \ldots, V_n]$.

$$
\begin{array}{rlll}
V ::= & B \mid R \mid A & & \text{Top-level values} \\
B ::= & \mathit{null} \mid \mathit{true} \mid \mathit{false} \mid n \mid s & & \text{Basic values} \\
R ::= & \mathit{ERec} \mid \mathit{Rec}(l, V, R) & & \text{Records} \\
A ::= & \mathit{EArr} \mid \mathit{Arr}(V, A) & & \text{Arrays}
\end{array}
$$

Semantics:

Records

$\mathit{Domain} : \mathit{FS}(\mathit{Keys} \times \mathit{Values})$

$$
\begin{array}{lll}
[\![\mathit{ERec}]\!] & \triangleq & \emptyset \\
[\![\mathit{Rec}(l, V, R)]\!] & \triangleq & \{(l, V)\} \cup [\![R]\!]
\end{array}
$$

Arrays

$\mathit{Domain} : \mathit{Lists}(\mathit{Values})$

$$
\begin{array}{lll}
[\![\mathit{EArr}]\!] & \triangleq & [\,] \\
[\![\mathit{Arr}(V, A)]\!] & \triangleq & [\![V]\!] :: A
\end{array}
$$

Notation:

$$
\begin{array}{lll}
\{l_1 : V_1, \ldots, l_n : V_n\} & \triangleq & \mathit{Rec}(l_1, V_1, \ldots \mathit{Rec}(l_n, V_n, \mathit{ERec})) \\
[V_1, \ldots, V_n] & \triangleq & \mathit{Arr}(V_1, \ldots \mathit{Arr}(V_n, \mathit{EArr}))
\end{array}
$$

Figure 2: Syntax of JSON data.

In JSON, a record is well-formed only if all its top-level keys are mutually different. In the sequel, we only consider well-formed JSON records, and we use Keys(R) to denote the set of the top-level keys of R.

Since a record is a set of fields, we identify two records that only differ in the order of their fields.

The syntax of the JSON schema language we adopt is depicted in Figure 3. The core of this language is captured by the non-terminals BT, RT, and AT which are a straightforward generalization of their B, R and A counterparts from the data model syntax.

As previously illustrated in Section 2, we adopt a very specific form of regular types in order to prepare an array type for fusion. Before fusion, an array type $[T_1, \ldots, T_n]$ is simplified as $[(T_1 + \ldots + T_n)*]$, or, more precisely, as $[\mathit{LFuse}(T_1, \ldots, T_n)*]$: instead of giving the content type element by element as in $[T_1, \ldots, T_n]$, we just say that it contains a sequence of values all belonging to $\mathit{LFuse}(T_1, \ldots, T_n)$ that will be defined as a compact super-type of $T_1 + \ldots + T_n$. This simplification is allowed by the fact that, besides the basic array types $AT = [T_1, \ldots, T_n]$, we also have the simplified array type $SAT = [T*]$, where T may be any type, including a union type.

A field $\mathit{OptRecT}(l, T, \ldots)$, represented as $l : T?$ in the simplified notation, represents an optional field, that is, a field that may be either present or absent in a record of the corresponding type. For example, a type $\{l : \mathtt{Num}?, m : (\mathtt{Str} + \mathtt{Null})\}$ describes records where l is optional and, if present, contains a number, while the m field is mandatory and may contain either *null* or a string.

A union type $T + U$ contains the union of the values from T and those from U. The empty type ϵ denotes the empty set.[1]

We define now schema semantics by means of the function $[\![\_]\!]$, defined as the minimal function mapping types to sets of values that satisfies the following equations. For the sake of simplicity we omit the case of basic types.

Auxiliary functions

$$
\begin{array}{lll}
S^0 & \triangleq & \{[\,]\} \\
S^{n+1} & \triangleq & \{[V] :: a \mid V \in S, a \in S^n[\,]\} \\
S^* & \triangleq & \bigcup_{i \in N} S^i
\end{array}
$$

Records

$\mathit{Domain} : \mathit{Sets}(\mathit{FS}(\mathit{Keys} \times \mathit{Values}))$

$$
\begin{array}{lll}
[\![\mathit{ERecT}]\!] & \triangleq & \{\emptyset\} \\
[\![\mathit{RecT}(l, T, RT)]\!] & \triangleq & \{\{(l, V)\} \cup R \mid V \in [\![T]\!],\ R \in [\![RT]\!]\} \\
[\![\mathit{OptRecT}(l, T, RT)]\!] & \triangleq & [\![\mathit{RecT}(l, T, RT)]\!] \cup [\![RT]\!]
\end{array}
$$

Arrays and **Simplified Arrays**

$\mathit{Domain} : \mathit{Sets}(\mathit{Lists}(\mathit{Values}))$

$$
\begin{array}{lll}
[\![\mathit{EArrT}]\!] & \triangleq & \{[\,]\} \\
[\![\mathit{ArrT}(T, AT)]\!] & \triangleq & \{[V] :: A \mid V \in [\![T]\!],\ A \in [\![AT]\!]\} \\
[\![[T*]]\!] & \triangleq & [\![T]\!]^*
\end{array}
$$

Union types

$$
\begin{array}{lll}
[\![\epsilon]\!] & \triangleq & \emptyset \\
[\![T + U]\!] & \triangleq & [\![T]\!] \cup [\![U]\!]
\end{array}
$$

The basic idea behind our type fusion mechanism is that we always generalize the union of two record types to one record type containing the keys of both, and similarly for the union of two array types. We express this idea as 'merging types that have the same kind'. The following $\mathit{kind}()$ function that maps each type to an integer ranging over $\{0, \ldots, 5\}$ is used to implement this approach.

$$
\begin{array}{lllllll}
\mathit{kind}(\mathtt{Null}) & = & 0 & \quad & \mathit{kind}(\mathtt{Str}) & = & 3 \\
\mathit{kind}(\mathtt{Bool}) & = & 1 & & \mathit{kind}(RT) & = & 4 \\
\mathit{kind}(\mathtt{Num}) & = & 2 & & \mathit{kind}(AT) = \mathit{kind}(SAT) & = & 5
\end{array}
$$

In the sequel, generic types are indicated by the metavariables T, U, W, while BT, RT, and AT are reserved for basic types, record types, and array types.

[1] The type ϵ is never used during type inference, since no value belongs to it. In greater detail, ϵ is actually a technical device that is only useful when an empty array type EArrT is simplified, before fusion, into a simplified array type: EArrT (that is, the type $[\,]$) is simplified as $[\epsilon*]$, which has the same semantics as EArrT, and our algorithms never insert ϵ in any other position.

$$\begin{array}{rcll} T & ::= & BT \mid RT \mid AT \mid SAT \mid \epsilon \mid T + T & \text{Top-level types} \\ BT & ::= & \mathtt{Null} \mid \mathtt{Bool} \mid \mathtt{Num} \mid \mathtt{Str} & \text{Basic types} \\ RT & ::= & ERecT \mid RecT(l, T, RT) \mid OptRecT(l, T, RT) & \text{Record types} \\ AT & ::= & EArrT \mid ArrT(T, AT) & \text{Array types} \\ SAT & ::= & [T*] & \text{Simplified array types} \end{array}$$

Notation:

$$\begin{array}{lcll} \{l_1 : T_1[?], \ldots, l_n : T_n[?]\} & \triangleq & [Opt]RecT(l_1, T_1, \ldots [Opt]RecT(l_n, T_n, ERecT)) & \textit{`?' is translated as `Opt'} \\ [\,] & \triangleq & EArrT & \\ [T_1, \ldots, T_n] & \triangleq & ArrT(T_1, \ldots ArrT(T_n, EArrT)) & \end{array}$$

Figure 3: Syntax of the JSON type language.

Later on, in order to express correctness of the fusion process we rely on the usual notion of subtyping (type inclusion).

Definition 4.1 (Subtyping) *Let T and U be two types. Then T is a subtype of U, denoted with $T <: U$, if and only if $[\![T]\!] \subseteq [\![U]\!]$.*

The subtyping relation is a partial order among types. We do not use any subtype checking algorithm in this work, but we exploit this notion to state properties of our schema inference approach.

5. SCHEMA INFERENCE

As already said, our approach is based on two steps: i) type inference for each single value in the input JSON data collection, and ii) fusion of types generated by the first step. We present these steps in the following two sections.

5.1 Initial Schema Inference

The first step of our approach consists of a Map phase that performs schema inference for each single value of the input collection. Type inference for single values is done according to the inference rules in Figure 4. Each rule allows one to infer the type of a value indicated in the conclusion (part below the line) in terms of types recursively determined in the premises (part above the line). Rules with no premises deal with the terminal cases of the recursive typing process, which infers the type of a value by simply reflecting the structure of the value itself. Note the particular case of record values where uniqueness of attribute keys l_i is checked. Also notice that these rules are deterministic: each possible value matches at most the conclusion of one rule. These rules, hence, directly define a recursive typing algorithm. The following lemma states soundness of value typing, and it can be proved by a simple induction.

Lemma 5.1 *For any JSON value V, $\vdash V \rightsquigarrow T$ implies $V \in [\![T]\!]$.*

It is worth noticing that schema inference done in this phase does not exploit the full expressivity of the schema language. Union types, optional fields, and repetition types (the Simplified Array Types) are never inferred, while these types will be produced by the schema fusion phase described next.

5.2 Schema Fusion

The second phase of our approach is meant to fuse all the types inferred in the first Map phase. The main mechanism of this phase is a binary fusion function, that is commutative and transitive. These properties are crucial as they ensure that the function can be iteratively applied over n types in a distributed and parallel fashion.

When fusion is applied over two types T and U, it outputs either a single type obtained by recursively merging T and U if they have the same kind, or the simple union $T + U$ otherwise. Since fusion may result in a union type, and since this is in turn fused with other types, possibly obtained by fusion itself, the fusion function has to deal with the case where union types $T = T_1 + \ldots + T_n$ and $U = U_1 + \ldots + U_m$ need to be fused. In this case, our fusion function identifies and fuses types T_j and U_h with matching kinds, while types of non-matching kinds are just moved unchanged into the output union type. As we will see later, the fusion process ensures the invariant property that in each output union type a given kind may occur at most once in each union; hence, in the two union types above, $n \leq 6$ and $m \leq 6$, since we only have six different kinds.

The auxiliary functions *KMatch* and *KUnmatch*, defined in Figure 5, respectively have the purpose of collecting pairs of types of the same kind in two union-types T_1 and T_2, and of collecting non-matching types. In Figure 5, two similar functions *FMatch* and *FUnmatch* are defined. They identify and collect fields having matching/unmatched keys in two input body record types RT_1 and RT_2.

These two functions are based on the auxiliary functions $\circ(T)$ and $\diamond(RT)$. The function $\circ(T)$ transforms a union type $T_1 + \ldots + T_n$ into the multiset of its *addends*, i.e non-union types $T_1, \ldots, T_n$. The function $\diamond(RT)$ transforms a record type $\{(l_1{:}T_1)^{\mathtt{m}_1}, \ldots (l_n{:}T_n)^{\mathtt{m}_n}\}$ into the set of its fields — in this case we can use a set since no repetition of keys is possible. Here we use $(l{:}T)^1$ to denote a mandatory field, $(l{:}T)^?$ to denote an optional field, and the symbols $\mathtt{m}$ and $\mathtt{n}$ for metavariables that range over $\{1, ?\}$.

We are now ready to present the fusion function. Its formal specification is given in Figure 6. We use the function $\oplus(S)$, that is a right inverse of $\circ(T)$ and rebuilds a union type from a multiset of non-union types, and the function $\bigcirc(S)$, that is a right inverse of $\diamond(RT)$ and rebuilds a record type from a set of fields. We also use $min(\mathtt{m}, \mathtt{n})$, which is a partial function that picks the "smallest" cardinality, by assuming $? < 1$.

The general case where types T_1 and T_2 that may be union types have to be fused is dealt with by the $Fuse(T_1, T_2)$

$$\textsc{(TypeNull)}\ \frac{}{\vdash null \rightsquigarrow \mathtt{Null}} \qquad \textsc{(TypeTrueBool)}\ \frac{}{\vdash \mathbf{true} \rightsquigarrow \mathtt{Bool}} \qquad \textsc{(TypeNumber)}\ \frac{}{\vdash n \rightsquigarrow \mathtt{Num}} \qquad \textsc{(TypeString)}\ \frac{}{\vdash s \rightsquigarrow \mathtt{Str}}$$

$$\textsc{(TypeEmptyRec)}\ \frac{}{\vdash ERec \rightsquigarrow ERecT} \qquad \textsc{(TypeEmptyArray)}\ \frac{}{\vdash EArr \rightsquigarrow EArrT}$$

$$\textsc{(TypeRec)}\ \frac{\vdash V \rightsquigarrow T \quad \vdash W \rightsquigarrow RT \quad l \notin \mathrm{Keys}(RT)}{\vdash Rec(l,V,W) \rightsquigarrow RecT(l,V,RT)} \qquad \textsc{(TypeArray)}\ \frac{\vdash V \rightsquigarrow T \quad \vdash W \rightsquigarrow AT}{\vdash Arr(V,W) \rightsquigarrow ArrT(T,AT)}$$

Figure 4: Type inference rules.

$\circ(T)$: *transforms a type into a multiset of non-union types, where* $\cup^b$ *is multiset union*

$$\begin{array}{lcll}
\circ(T_1+T_2) & := & \circ(T_1)\cup^b \circ(T_2) & \\
\circ(\epsilon) & := & \{\ \} & \\
\circ(T) & := & \{T\} & \text{when } T\neq T_1+T_2 \text{ and } T\neq\epsilon \\
KMatch(T_1,T_2) & := & \{(U_1,U_2)\mid U_1\in\circ(T_1), U_2\in\circ(T_2), kind(U_1)=kind(U_2)\} & \\
KUnmatch(T_1,T_2) & := & \{U_1\in\circ(T_1)\mid \forall U_2\in T_2.\ kind(U_1)\neq kind(U_2)\} & \\
 & & \cup\{U_2\in\circ(T_2)\mid \forall U_1\in\circ(T_1).\ \ kind(U_2)\neq kind(U_1)\} &
\end{array}$$

$\diamond(RT)$: *transforms a record type into a set of fields*

$$\begin{array}{lcl}
\diamond(ERecT) & := & \emptyset \\
\diamond(RecT(l,T,RT)) & := & \{(l{:}T)^{1}\}\cup\diamond(RT) \\
\diamond(OptRecT(l,T,RT)) & := & \{(l{:}T)^{?}\}\cup\diamond(RT) \\
FMatch(RT_1,RT_2) & := & \{((l{:}T)^{\mathtt{n}},(k{:}U)^{\mathtt{m}})\mid (l{:}T)^{\mathtt{n}}\in\diamond(RT_1) \text{ and } (k{:}U)^{\mathtt{m}}\in\diamond(RT_2) \text{ and }\ l=k\} \\
FUnmatch(RT_1,RT_2) & := & \{(l{:}T)^{\mathtt{n}}\in\diamond(RT_1)\mid \forall (k{:}U)^{\mathtt{m}}\in\diamond(RT_2).\ l\neq k\}\cup\{(l{:}T)^{\mathtt{n}}\in\diamond(RT_2)\mid \forall (k{:}U)^{\mathtt{m}}\in\diamond(RT_1).\ l\neq k\}
\end{array}$$

Figure 5: Auxiliary functions.

function. According to what was said before, it recursively applies *LFuse* to pairs of types coming from T_1 and T_2 and having the same kind, while unmatched types are simply returned in the output union type.

The specification of *LFuse* is captured by lines 2 to 7. Line 2 deals with the case where the input types are two identical basic types. In this case, the fusion yields the input basic type. Line 3 deals with the case where the input types are records. In this case, pairs of fields whose keys match are recursively fused by calling *LFuse*, the lowest cardinality is chosen for each, so that a field is mandatory only if is mandatory in both record types, whereas the unmatching fields are copied in the result type as optional fields.

The remaining lines of *LFuse* are dedicated to the case where the input types are arrays. Each of these lines deals with a combination among original and simplified arrays by ensuring that *Fuse* is called over the body types of arrays that have been simplified through the call of *collapse*. While line 4 faces the case that the two types have not been subject to fusion yet, lines 5-7 deal with the case that one of the input is the result of previous fusion operations, and therefore it has a *-expression as a body (recall the discussion in Section 2). Lines 8 and 9 are dedicated to the array simplification function *collapse*. This function simply relies on *Fuse* in order to generate an over-approximation of all the different types that are found in the original array type, in order to prepare the array type for the fusion process.

To illustrate both body array type simplification and record fusion, consider the following type T:

$$\begin{array}{rcl} T & = & [\mathtt{Num},\mathtt{Bool},\mathtt{Num},\ \{l_1:\mathtt{Num}, l_2:\mathtt{Str}\},\ \{l_1:\mathtt{Num}\}, \\ & & \quad \{l_2:\mathtt{Bool}, l_3:\mathtt{Str}\}] \end{array}$$

We have that $collapse(T)$ is equal to:

$$(\mathtt{Num}+\mathtt{Bool}+\{l_1:\mathtt{Num}, l_2:\mathtt{Str}+\mathtt{Bool}, (l3:\mathtt{Str})?\})$$

Note that only one record type is created, by iterating fusion over the three record types. Also note that there is a good level of size reduction entailed by simplification. This happens in the most frequent cases (where elements of an array share most of their structure), while size reduction becomes weaker when very heterogeneous records appear in the array body type (in the particular case where no field key is shared among records, the unique record type given by simplification contains all keys, with their associated types, as optional fields).

To conclude this section, the following theorems state the main theoretical properties of the fusion process: correctness, commutativity and associativity. The crucial role played by these properties has already been discussed in the previous sections.

All these properties hold for types that respect the invariant that types of a given kind can occur at most once in each union. We use the term "normal types" to refer to such types. All of our algorithms respect this invariant, that is, they only generate normal types.

We first deal with correctness.

$\oplus(S)$: *transforms a multiset of addends into a union type of these addends, right inverse for* $\circ(T)$

$$\begin{array}{lll}
\oplus(\{\ \}) & := & \epsilon \\
\oplus(\{T\}) & := & T \\
\oplus(\{T_1, T_2, \ldots, T_n\}) & := & T_1 + \oplus(\{T_2, \ldots, T_n\}) \qquad \text{when } n \geq 2
\end{array}$$

$\bigcirc(S)$: *transforms a set of fields into a record type, right inverse for* $\diamond(RT)$

$$\begin{array}{lll}
\bigcirc(\emptyset) & := & \mathit{ERecT} \\
\bigcirc(\{(l{:}T)^1\} \cup S) & := & \mathit{RecT}(l, T, \bigcirc(S)) \\
\bigcirc(\{(l{:}T)^?\} \cup S) & := & \mathit{OptRecT}(l, T, \bigcirc(S))
\end{array}$$

$$\begin{array}{lll}
1.\ \mathit{Fuse}(T_1, T_2) & := & \oplus(\{\mathit{LFuse}(U_1, U_2) \mid (U_1, U_2) \in \mathit{KM}\} \cup^b \{U_3 \mid U_3 \in \mathit{KU}\}) \\
 & & \text{ with } \mathit{KM} = \mathit{KMatch}(T_1, T_2),\ \mathit{KU} = \mathit{KUnmatch}(T_1, T_2) \\
 & & \\
2.\ \mathit{LFuse}(B, B) & := & B \ \text{ with } \mathit{kind}(B) < 4 \\
 & & \\
3.\ \mathit{LFuse}(RT_1, RT_2) & := & \bigcirc(\{l{:}\mathit{Fuse}(T_1, T_2)^{\mathit{min}(\mathtt{m},\mathtt{n})} \mid ((l{:}T_1)^{\mathtt{m}}, (l{:}T_2)^{\mathtt{n}}) \in \mathit{FM}\} \\
 & & \quad \cup \{(l{:}T)^? \mid (l{:}T)^{\mathtt{m}} \in \mathit{FU}\}) \\
 & & \text{ with } \mathit{FM} = \mathit{FMatch}(RT_1, RT_2),\ \mathit{FU} = \mathit{FUnmatch}(RT_1, RT_2) \\
 & & \\
4.\ \mathit{LFuse}(AT_1, AT_2) & := & [\,\mathit{Fuse}(\mathit{collapse}(AT_1), \mathit{collapse}(AT_2))*\,] \\
5.\ \mathit{LFuse}([T*], AT) & := & [\,\mathit{Fuse}(T, \mathit{collapse}(AT))*\,] \\
6.\ \mathit{LFuse}(AT, [T*]) & := & [\,\mathit{Fuse}(\mathit{collapse}(AT), T)*\,] \\
7.\ \mathit{LFuse}([T_1*], [T_2*]) & := & [\,\mathit{Fuse}(T_1, T_2)*\,] \\
 & & \\
8.\ \mathit{collapse}(\mathit{EArrT}) & := & \epsilon \\
9.\ \mathit{collapse}(\mathit{ArrT}(T, AT)) & := & \mathit{Fuse}(T, \mathit{collapse}(AT))
\end{array}$$

Figure 6: The formal specification of the type fusion.

Theorem 5.2 (Correctness of ***Fuse*****)** *Given two normal types T_1 and T_2, if $T_3 = \mathit{Fuse}(T_1, T_2)$, then $T_1 <: T_3$ and $T_2 <: T_3$.*

The proof of the above theorem relies on the following lemma.

Lemma 5.3 (Correctness of ***LFuse*****)** *Given two non-union normal types T_1 and T_2 with the same kind, we have that $T_3 = \mathit{LFuse}(T_1, T_2)$ implies both $T_1 <: T_3$ and $T_2 <: T_3$.*

Another important property of fusion is commutativity.

Theorem 5.4 (Commutativity) *The following two properties hold.*

1. *Given two normal types T_1, T_2, we have $\mathit{Fuse}(T_1, T_2) = \mathit{Fuse}(T_2, T_1)$.*
2. *Given two non-union normal types T and U having the same kind, we have $\mathit{LFuse}(T, U) = \mathit{LFuse}(U, T)$.*

Associativity of binary type fusion is stated by the following theorem.

Theorem 5.5 (Associativity) *The following two properties hold.*

1. *Given three normal types T_1, T_2, and T_3, we have*

$$\mathit{Fuse}(\mathit{Fuse}(T_1, T_2), T_3) = \mathit{Fuse}(T_1, \mathit{Fuse}(T_2, T_3))$$

2. *Given three non-union normal types T, U and V of the same kind, we have*

$$\mathit{LFuse}(\mathit{LFuse}(T, U), V) = \mathit{LFuse}(T, \mathit{LFuse}(U, V))$$

6. EXPERIMENTAL EVALUATION

In this section we present an experimental evaluation of our approach whose main goal is to validate our precision and succinctness claims. We also incorporate a preliminary study on using our approach in a cluster-based environment for the sake of dealing with complex large datasets.

6.1 Experimental Setup and Datasets

For our experiments, we used Apache Spark 1.6.1 [7] installed on two kinds of hardware. The first configuration consists in a single Mac mini machine equipped with an Intel dual core 2.6 Ghz processor, 16GB of RAM, and a SATA hard-drive. This machine is mainly used for verifying the precision and succinctness claims. In order to assess the scalability of our approach and its ability to deal with large datasets, we also exploited a small size cluster of six nodes connected using a Gigabit link with 1Gb speed. Each node is equipped with two 10-core Intel 2.2 Ghz CPUs, 64GB of RAM, and a standard RAID hard-drive.

The choice of using Spark is intuitively motivated by its widespread use as a platform for processing large datasets of different kinds (e.g., relational, semi-structured, and graph data). Its main characteristic lies in its ability to keep large datasets into main-memory in order to process them in a fast and efficient manner. Spark offers APIs for major programming languages like Java, Scala, and Python. In particular, Scala serves our case well since it makes the encoding of pattern matching and inductive definitions very easy. Using Scala has, for instance, allowed us to implement both the type inference and the type fusion algorithms in a rather straightforward manner starting from their respective formal specifications.

The type inference implementation extends the Json4s library [4] for parsing the input JSON documents. This library yields a specific Scala object for each JSON construct (array, record, string, etc), and this object is used by our implementation to generate the corresponding type construct. The type fusion implementation follows a standard functional programming approach and does not need to be commented.

It is important to mention that the Spark API offers a feature for extracting a schema from a JSON document. However, this schema inference suffers from two main drawbacks. First, the inferred schemas do not contain regular expressions, which prevents one from concisely representing repeated types, while our type system uses the Kleene-Star to encode the repetition of types. Second, the Spark schema extraction is imprecise when it comes to deal with arrays containing *mixed content*, such as, for instance, an array of the form:

$$[\mathtt{Num}, \mathtt{Str}, \{l : \mathtt{Str}\}]$$

In such a case, the Spark API uses *type coercion* yielding an array of type String only. In our case, we can exploit union types to generate a much more precise type:

$$[(\mathtt{Num} + \mathtt{Str} + \{l : \mathtt{Str}\})*]$$

For our experiments we used four datasets. The first two datasets are borrowed from an existing work [13] and correspond to data crawled from GitHub and from Twitter. The third dataset consists in a snapshot of Wikidata [6], a large repository of facts feeding the Wikipedia portal. The last dataset consists in a crawl of NYTimes articles using the NYTimes API [5]. A detailed description of each dataset is provided in the sequel.

GitHub.

This dataset corresponds to metadata generated upon pull requests issued by users willing to commit a new version of code. It comprises 1 million JSON objects sharing the same top-level schema and only varying in their lower-level schema. All objects of this dataset consist exclusively of records, sometimes nested, with a nesting depth never greater than four. Arrays are not used at all.

Twitter.

Our second dataset corresponds to metadata that are attached to the tweets shared by Twitter users. It comprises nearly 10 million records corresponding, in majority, to tweet entities. A tiny fraction of these records corresponds to a specific API call meant to delete tweets using their ids. This dataset is interesting for our experiment for many reasons. First, it uses both records and arrays of records, although the maximum level of nesting is 3. Second, it contains five different top-level schemas sharing common parts. Finally, it mixes two kinds of JSON records (tweets and deletes). This dataset is useful to assess the effectiveness of our typing approach when dealing with arrays.

Wikidata.

The largest dataset comprises 21 million records reaching a size of 75GB and corresponding to Wikipedia *facts*. These facts are structured following a fixed schema, but suffer from a poor design compared to the previous datasets. For instance, an important portion of Wikidata objects corresponds to *claims* issued by users. These user identifiers are directly encoded as keys, whereas a clean design would suggest encoding this information as a value of a specific key called *id*, for example. This dataset can be of interest to our experiments since several records reach a nesting level of 6.

NYTimes.

The last dataset we are considering here is probably the most interesting one and comprises approximately 1.2 million records and reaches the size of 22GB. Its objects feature both nested records and arrays, and are nested up to 7 levels. Most of the fields in records are associated to text data, which explains the large size of this dataset compared to the previous ones. These records encode metadata about news articles, such as the headline, the most prominent keywords, the lead paragraph as well as a snippet of the article itself. The interest of this dataset lies in the fact that the content of fields is not fixed and varies from one record to another. A quick examination of an excerpt of this dataset has revealed that the content of the *headline* field is associated, in some records, to subfields labeled *main*, *content_kicker*, *kicker*, while in other records it is associated to subfields labeled *main* and *print_headlines*. Another common pattern in this dataset is the use of *Num* and *Str* types for the same field.

In order to compare the results of our experiments using the four datasets, we decided to limit the size of every dataset to the first million records (the size of the smallest one). We also created, starting from each dataset, subdatasets by restricting the original ones to respectively thousand (1K), ten thousands (10K) and one hundred thousands (100K) records chosen in a random fashion. Table 1 reports the size of each of these sub-datasets.

	1K	10K	100K	1M
GitHub	14MB	137MB	1.3GB	14GB
Twitter	2.2MB	22 MB	216MB	2.1GB
Wikidata	23MB	155MB	1.1GB	5.4GB
NYTimes	10MB	189MB	2GB	22GB

Table 1: (Sub-)datasets sizes.

6.2 Testing Scenario and Results

The main goal of our experiments is to assess the effectiveness of our approach and, in particular, to understand if it is able to return succinct yet precise fused types. To do so we report in Tables 2 to 5, for each dataset, the number of distinct types, the min, max, and average size of these types as well as the size of the fused type. The notion of size of a type is standard, and corresponds to the size (number of nodes) of its Abstract Syntax Tree. For fairness, one can consider the average size as a baseline wrt which we compare the size of the fused type. This helps us judge the effectiveness of our fusion at collapsing common parts of the input types.

From Tables 2, 3, and 4, it is easy to observe that our primary goal of succinctness is achieved for the GitHub and the Twitter datasets. Indeed, the ratio between the size of the fused type and that of the average size of the input types is not bigger than 1.4 for GitHub whereas it is bounded by 4 for Twitter, which are relatively good factors. These results are not surprising: GitHub objects are homogeneous. Twitter has a more varying structure and, in addition, it mixes two different kinds of objects that are deletes and tweets, as outlined in the description of this dataset. This explains the slight difference in terms of compaction wrt GitHub. As expected, the results for Wikidata are worse than the results for the previous datasets, due to the particularity of this dataset concerning the encoding of user-ids as keys. This has an impact on our fusion technique, which relies on keys to merge the underlying records. Still, our fusion algorithm manages to collapse the common parts of the input types as testified by the fact that the size of the fused types is smaller than the sum of the input types.[2] Finally, the results for NYtimes dataset, which features many irregularities, are promising and even better than the rest. This can be explained by the fact that the fields in the first level are fixed while the lower level fields may vary. This does not happen in the previous datasets, where the variations occur on the first level.

	Inferred types size				Fused
	# types	min.	max.	avg.	type size
1K	29	147	305	233	321
10K	66	147	305	239	322
100K	261	147	305	246	330
1M	3,043	147	319	257	354

Table 2: Results for GitHub.

	Inferred types size				Fused
	# types	min.	max.	avg.	type size
1K	167	7	218	74	221
10K	677	7	276	75	273
100K	2,320	7	308	75	277
1M	8,117	7	390	77	299

Table 3: Results for Twitter.

	Inferred types size				Fused
	# types	min.	max.	avg.	type size
1K	999	27	36,748	1,215	37,258
10K	9,886	21	36,748	866	82,191
100K	95,298	11	39,292	607	87,290
1M	640,010	11	39,292	310	117,010

Table 4: Results for Wikidata.

Execution times for the type inference and the type fusion for GitHub, Twitter, and Wikidata datasets are reported in Table 6. As it can be observed, processing the Wikidata dataset is more time-consuming than processing the two other datasets. This is explained, once again, by the nature of the Wikidata dataset. Observe also that the processing time of GitHub is larger than that of Twitter due to the size of the former dataset that is larger than the latter one.

	Inferred types size				Fused
	# types	min.	max.	avg.	type size
1K	555	299	887	597.25	88
10K	2,891	6	943	640	331
100K	15,959	6	997	755	481
1M	312,458	6	1,046	674	760

Table 5: Results for NYTimes.

	1K	10K	100K	1M
GitHub	1s	4s	32s	297s
Twitter	0	1s	7s	73s
Wikidata	7s	15s	121s	925s

Table 6: Typing execution times.

6.3 Scalability

To assess the scalability of our approach, we have deployed the typing and the fusion implementations on our cluster. To exploit the full capacity of the cluster in terms of number of cores, we set the number of cores to 120, that is, 20 cores per node. We also assign to our job 300GB of main memory, hence leaving 72GB for the task manager and other runtime monitoring processes. We used the NYTimes full dataset (22GB) stored on HDFS. Because our approach requires two steps (type inference and type fusion), we adopted a strategy where the results of the type inference step are persisted into main-memory to be directly available to the fusion step. We ran the experiments on datasets of varying size obtained by restricting the full one to the first fifty, two hundred-fifty and five hundred thousands records, respectively. The results for these experiments are reported in Table 7 together with some statistics on these datasets (number of records and cardinality of the distinct types). It can be observed that execution time increases linearly with the dataset size.

size	# records	# distinct types	time
1GB	50,000	5,679	2 min
4.5GB	250,000	54,868	4.4 min
9GB	500,000	128,943	8.5 min
22GB	1,184,943	312,458	12.5 min

Table 7: Scalability - NYTimes dataset.

In an attempt to optimize the execution time on the cluster, we started by analyzing the execution and realized that the full capacity of the cluster was not exploited. Indeed, the HDFS uses only one node to store the entire dataset, which does not allow the parallelism to be exploited. We also observed that the intermediate results produced by the type inference step were split on only two nodes. The overall effect is that the computation was performed on two nodes while the remaining four nodes were idle.

[2] The total size of input types can be roughly estimated by multiplying either the minimum, maximum, or average size with the number of types.

To overcome this problem, we considered a strategy based on partitioning the input data that would force Spark to take full advantage of the cluster. In order to avoid the overhead of data shuffling, the ideal solution would be to force computation to be local until the end of the processing. Because Spark 1.6 does not explicitly allow such an option, we had to opt for a manual strategy where each partition of data is processed in isolation, and each of the inferred schema is finally fused with the others (this is a fast operation as each schema to fuse has a very small size). The purpose is to simulate the realistic situation where Spark processes data exclusively locally, thus avoiding the overhead of synchronization. The times for processing each partition are reported in Table 8. The average time is 2.85 minutes, which is a rather reasonable time for processing a dataset of 22 GB.

	# objects	# types	time
partition 1	284,943	67,632	2.4 min
partition 2	300,000	83,226	3.8 min
partition 3	300,000	89,929	1.9 min
partition 4	300,000	84,333	3.3 min

Table 8: Partition-based processing of NYTimes.

Note that this simple yet effective optimization is possible thanks to the associativity of our fusion process.

7. CONCLUSIONS AND FUTURE WORK

The approach described in this paper is a first step towards the definition of a schema-based mechanism for exploring massive JSON datasets. This issue is of great importance due to the overwhelming quantity of JSON data manipulated on the web and due to the flexibility offered by the systems managing these data.

The main idea of our approach is to infer schemas for the input datasets in order to get insights about the structure of the underlying data; these schemas are succinct yet precise, and faithfully capture the structure of the input data. To this end, we started by identifying a schema language with the operators needed to ensure succinctness and precision of our inferred schemas. We, then, proposed a fusion mechanism able to detect and collapse common parts of the input types. An experimental evaluation on several datasets validated our claims and showed that our type fusion approach actually achieves the goals of succinctness, precision, and efficiency.

Another benefit of our approach is its ability to perform type inference in an incremental fashion. This is possible because the core of our technique, fusion, is incremental by essence. One possible and interesting application would be to process a subset of a large dataset to get a first insight on the structure of the data before deciding whether to refine this partial schema by processing additional data.

In the near future we plan to enrich schemas with statistical and provenance information about the input data. Furthermore, we want to improve the precision of the inference process for arrays and study the relationship between precision and efficiency.

8. ACKNOWLEDGMENTS

Houssem Ben Lahmar has been partially supported by the project D03 of SFB/Transregio 161.[3]

9. REFERENCES

[1] The JSON Query Language. `http://www.jsoniq.org`.

[2] Json schema definition language. `http://jsoniq.org/docs/JSound/html-single/`.

[3] Json schema language. `http://json-schema.org`.

[4] Json4s library. `http://json4s.org`.

[5] Nytimes api. `https://developer.nytimes.com/`.

[6] Wikidata. `https://dumps.wikimedia.org/wikidatawiki/entities/`.

[7] Apache Spark. Technical report, 2016. `http://spark.apache.org`.

[8] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyên. Type-based xml projection. VLDB '06, pages 271–282, 2006.

[9] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011.

[10] T. Bray. The javascript object notation (JSON) data interchange format, 2014.

[11] S. Cebiric, F. Goasdoué, and I. Manolescu. Query-oriented summarization of RDF graphs. *PVLDB*, 8(12):2012–2015, 2015.

[12] D. Colazzo, G. Ghelli, and C. Sartiani. Typing massive json datasets. In *XLDI '12, Affiliated with ICFP*, 2012.

[13] M. DiScala and D. J. Abadi. Automatic generation of normalized relational schemas from nested key-value data. SIGMOD '16, pages 295–310, 2016.

[14] D. D. Freydenberger and T. Kötzing. Fast learning of restricted regular expressions and dtds. *Theory Comput. Syst.*, 57(4):1114–1158, 2015.

[15] Z. H. Liu, B. Hammerschmidt, and D. McMahon. Json data management: Supporting schema-less development in rdbms. SIGMOD '14, pages 1247–1258, 2014.

[16] J. McHugh and J. Widom. Query optimization for xml. VLDB '99, pages 315–326. Morgan Kaufmann Publishers Inc., 1999.

[17] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Internet Technol.*, 5(4):660–704, Nov. 2005.

[18] S. Nestorov, S. Abiteboul, and R. Motwani. Infering structure in semistructured data. *SIGMOD Record*, 26(4):39–43, 1997.

[19] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In L. M. Haas and A. Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.*, pages 295–306. ACM Press, 1998.

[20] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. Foundations of json schema. WWW '16, pages 263–273, 2016.

[21] S. Scherzinger, E. C. de Almeida, T. Cerqueus, L. B. de Almeida, and P. Holanda. Finding and fixing type

[3] http://www.trr161.de/interfak/forschergruppen/sfbtrr161

mismatches in the evolution of object-nosql mappings. In T. Palpanas and K. Stefanidis, editors, *Proceedings of the Workshops of the EDBT/ICDT 2016*, volume 1558 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[22] L. Wang, S. Zhang, J. Shi, L. Jiao, O. Hassanzadeh, J. Zou, and C. Wangz. Schema management for document stores. *Proc. VLDB Endow.*, 8(9):922–933, May 2015.

Distributed in-memory SPARQL Processing via DOF Analysis

Roberto De Virgilio
Roma Tre University, Rome, Italy
dvr@dia.uniroma3.it

ABSTRACT

Using so-called *triple patterns* as building blocks, SPARQL queries search for specified patterns in RDF data. Although many aspects of the challenges faced in large-scale RDF data management have already been studied in the database research community, current approaches provide centralized DBMS (or disk) based solutions, with high consumption of resources; moreover, these exhibit very limited flexibility dealing with queries, at various levels of granularity and complexity (*e.g.*, SPARQL queries involving UNION or OPTIONAL operators). In this paper we propose a computational in-memory framework for distributed SPARQL query answering, based on the notion of *degree of freedom* of a triple. This algorithm relies on a general model of RDF graph based on the first principles of linear algebra, in particular on *tensorial calculus*. Experimental results show that our approach, utilizing linear algebra techniques can process analysis efficiently, when compared to recent approaches.

CCS Concepts

•Information systems → Data management systems; World Wide Web; •Computing methodologies → Linear algebra algorithms; Distributed algorithms;

Keywords

RDF; SPARQL; Tensor Calculus

1. INTRODUCTION

Today, many organizations and practitioners are all contributing to the "Web of Data", building RDF repositories of huge amounts of semantic data, posing serious challenges in maintaining and querying large datasets. Modern scenarios involve analyses of very large semantic datasets, usually employing the SPARQL query language. Many aspects of large-scale RDF data management have already been studied in the database research community, including native RDF storage layout and index structures [18], SPARQL query processing and optimization [8], as well as formal semantics and computational complexity of SPARQL [20, 23].

Challenges. Examining the prevalent trend in semantic information storage and inspection, we face two major challenges: management of large datasets, and scalability of both storage and querying. As size increases, both storage and analysis must scale accordingly; however, despite major efforts, building performant and scalable RDF systems is still a hurdle. Most of the current state-of-the-art approaches consider SPARQL as *the SQL for RDF*, and therefore they usually employ RDBMS-based solutions to store RDF graphs, and to execute a SPARQL query through SQL engines (e.g. Jena or Sesame). Moreover, popular systems are developed as single-machine tools [18, 28], which hinder performances as the size of RDF dataset continues to escalate. In particular Jena, Sesame, RDF-3X [18], BitMat [1], TripleBit [29] and GADDI [31] represent centralized approaches exploiting single-machine implementations of edge (e.g., [18, 1]) and subgraph (e.g. [31]) index based approaches to graph matching over graph-shaped data (e.g. RDF). In this context efficiency is driven by various forms of horizontal and vertical partitioning scheme [4], or by sophisticated encoding schemes [1, 29]. Hence, such proposals require the replication of data in order to improve performances, or introduces several indexing functions that increase the overall size of stored information. Lately, some distributed RDF processing systems have been developed [30, 8, 19, 9]. Trinity.RDF [30] is a distributed GraphDB engine for RDF graph matching: it observes that query processing on RDF graphs (i.e. by SPARQL) requires many graph operations not having locality [23], but relies exclusively on random accesses. Therefore typical disk-based triple-store solutions are not feasible for performing fast random accesses on hard disks. To this aim, among distributed approaches, Trinity.RDF exploits GraphDB technology to store RDF data in a native form and implements a scheduling algorithm to reduce step-by-step the amount of data to analyze during SPARQL query execution. However non-selective queries require many parallel join executions that the generic architecture of Trinity.RDF is not able to integrate, as it is common in MapReduce approaches using Hadoop (e.g., [11]). On the other hand, MapReduce solutions involve a non-negligible overhead, due to the synchronous communication protocols and hob scheduling strategies. Therefore, H2RDF+ [19] builds eight indexes using HBase. It uses Hadoop to perform sort-merge joins during query processing. DREAM [9] proposes the Quadrant-IV paradigm and partitions queries instead of data and selects different number of machines to execute different SPARQL queries based on their complexity. It employs a graph-based query planner and a cost model to outperform its competitors. In addition, we mention the TriAD distributed system [8], embedding a main-memory architecture, based on the master-slave paradigm. The problem of such system, as in other approaches and as it will be shown experimentally, is that it exhibits complex indexing (i.e., SPO permutation indexing) and

Series ISSN: 2367-2005

10.5441/002/edbt.2017.22

partition schemes (i.e. RDF summary graph), damaging seriously the maintenance of the approach itself and making not possible to exploit completely an in-memory (distributed) engine. Moreover graph data are often maintained in a relational store which is replicated on the disk of each of the underlying nodes of a cluster; managing big attributed graphs on a single machine may be infeasible, especially when the machine's memory is dwarfed by the size of the graph topology.

Contribution. In this paper we propose a novel distributed in-memory approach for SPARQL query processing on *highly unstable* very large datasets. Our objective is to provide a performance-oriented system able to analyze RDF graphs on which *no a priori knowledge* is possible or available, and to avoid collection (exploitation) of complex statistics on initial data and/or frequent past queries. Based on the notion of DOF, the *degree of freedom* of a triple pattern, that is a measure of triple pattern's explicit constraints, we rely on a simple and optimal scheduling algorithm that builds incrementally answers to a SPARQL query. Intuitively, a pattern with no constraints, *i.e.*, constituted only by variables, has the highest DOF, while one constituted by only constants is associated with the lowest DOF. Specifically, our scheduling starts from triple patterns with the lowest degree of freedom, and proceeds in bounding variables incrementally to their values by selecting the triple pattern with the *highest probability of decreasing the search-space*.

As opposed to DBMS-based (single-machine) approaches, our approach avoids any schema or indexing definition over the RDF graph to query, being reindexing impractical for both space and time consumption in a highly volatile environment. Additionally, we highlight as, in a distributed query system, storing RDF data in disk-based triple stores hinders performances, as queries on such graphs are non-local [30], and therefore random access techniques are required to speedup processing. We define a general model of RDF graph based on first principles derived from the *tensor algebra* field. Many real-world data (e.g., knowledge bases, web data, network traffic data, and many others [25, 13, 5]) with multiple attributes are represented as multi-dimensional arrays, called tensors. In analyzing a tensor, tensor decompositions are powerful tools in many data mining applications: correlation analysis on sensor streams [25], latent semantic indexing on DBLP publication data [26], multi-aspect forensics on network data [16], network discovery [5] and so on.

Leveraging such background, this paper proposes a formal tensor representation and endowed with specific operators allowing to perform efficiently our scheduling algorithm for both quick decentralized and centralized massive analysis on large volumes of data—*i.e.*, billions of triples. We strongly rely on an *in-memory distributed approach*, so that our framework may comfortably analyze any given RDF dataset, without any cumbersome processing; in other words, the tensor construction itself is the only processing operation we perform. Our model and operations, inherited by linear algebra and tensor calculus, are therefore theoretically sound, and their implementation exploit the underlying hardware for searching in the solution space. In detail, by applying bit-oriented operations on data, each computational node in our distributed system is able to exploit low-level CPU operations, *e.g.*, 64-bit or 128-bit x86 register opcodes; additionally, operations are carried out in a *cache-oblivious* manner, thus taking advantage of both L1 and L2 caches on modern architectures. Due to the properties of our tensorial model, we are able to dissect tensors (*i.e.*, $\mathcal{R}_i$) representing RDF graphs into several chunks to be processed independently (*i.e.*, by each process p_i), as shown in Figure 1.

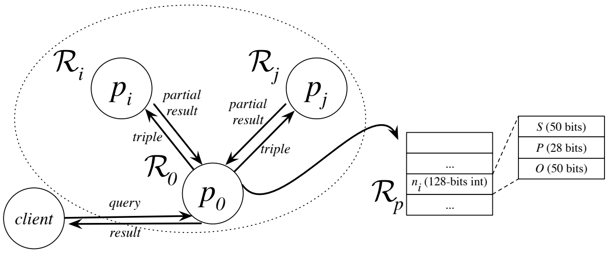


Figure 1: Distributed query processing in TENSORRDF.

Outline. Our manuscript is organized as follows. Section 2 will introduce our general model of a RDF graph, accompanied by a formal tensorial representation, subsequently put into practice in Section 3, where we provide a method for RDF data analysis. Section 4 describes a scheduling algorithm to perform SPARQL queries. Section 5 illustrates the physical modeling of our framework, while Section 6 discusses the complexity of all involved operations. We benchmark our approach with several test beds, and supply the results in Section 7, with the available literature discussed in Section 8. Finally Section 9 sketches some conclusions and future work.

2. RDF AND SPARQL MODELING

This section is devoted to give a rigorous definition of an ontology, with respect to RDF and Semantic Web.

RDF. Data in RDF is built from three disjoints sets $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{L}$ of *IRIs*, *blank nodes*, and *literals*, respectively. All information in RDF is represented by triples of the form $\langle s, p, o \rangle$, where s is called the *subject*, p is called the *predicate*, and o is called the *object*. To be valid, it is required that $s \in \mathcal{I} \cup \mathcal{B}$; $p \in \mathcal{I}$; and $o \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$. RDF is a representation of an ontology. An ontology may be viewed as a set of relations between objects, or more in general, as a *function* that, given two entities s and o, and a relation p, returns a truth value corresponding to the condition of whether or not the two entities are related.

DEFINITION 1 (ONTOLOGY TENSOR). *Let $\mathcal{S}$ be the finite set of* subjects, *$\mathcal{O}$ the finite set of* objects, *and $\mathcal{P}$ the finite set of* predicates, *the* ontology tensor *is a rank-3 tensor $\mathcal{T} : \mathcal{S} \times \mathcal{P} \times \mathcal{O} \longrightarrow \mathbb{B}$, being $\mathbb{B}$ a boolean ring.*

This general definition of ontology tensor must be related to existing representations, in particular with RDF. Hence, we can introduce a direct mapping between the sets $\mathcal{I}$, $\mathcal{B}$, $\mathcal{L}$ and the above definition:

DEFINITION 2 (RDF SETS). *The finite sets $\mathcal{S}$, $\mathcal{P}$, and $\mathcal{O}$ are defined as follows: $\mathcal{S} := \mathcal{I} \cup \mathcal{B}$, $\mathcal{P} := \mathcal{I}$, and $\mathcal{O} := \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$.*

Let us briefly focus on the fact that, by definition, $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{L}$ are finite and *countable*. Therefore, their union is a countable set, and so $\mathcal{S}$, $\mathcal{P}$, and $\mathcal{O}$, consequently. The countability property makes it possible to relate each set to $\mathbb{N}$ via an injective function, *i.e.*, we can "order" all the elements.

DEFINITION 3 (RDF SET INDEXING). *Given the finite countable RDF sets $\mathcal{S}$, $\mathcal{P}$, and $\mathcal{O}$, we introduce their respective* indexing *functions $\mathbb{S}$, $\mathbb{P}$, and $\mathbb{O}$ of* subjects, predicates, *and* objects*: $\mathbb{S} : \mathcal{S} \longrightarrow \mathbb{N}$, $\mathbb{P} : \mathcal{P} \longrightarrow \mathbb{N}$, and $\mathbb{O} : \mathcal{O} \longrightarrow \mathbb{N}$.*

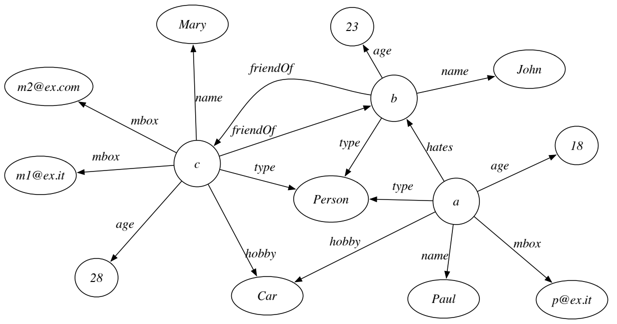


Figure 2: An example of RDF graph G.

The introduced functions, given the finiteness and countability properties are not only injective, but also surjective, *i.e.*, we introduced a bijection between a subset of $\mathbb{N}$, and $\mathcal{S}$, $\mathcal{P}$, $\mathcal{O}$. The *RDF set indexing* functions map elements of RDF sets to a (subset of) natural numbers. Let us focus on the ontology graph G given in Figure 2, constituted of 14 nodes (*i.e.*, 4 resources and 10 literals) and 7 properties. In this case, we have $\mathbb{S}(a)$ = 1, $\mathbb{S}(b)$ = 2 $\mathbb{S}(c)$ = 3, $\mathbb{P}(age)$ = 1, $\mathbb{P}(friendOf)$ = 2, and so on. Their inverse functions are, being bijections, well defined, *e.g.*, $\mathbb{S}^{-1}(3)$ = *c*.

DEFINITION 4 (RDF TENSOR). *Let G be a RDF graph. The RDF tensor $\mathcal{R}(G) =: \mathcal{R}$ on G is an ontology tensor such that*

$$\mathcal{R} = (r_{ijk}) := \begin{cases} 1, & \langle \mathbb{S}^{-1}(i), \mathbb{P}^{-1}(j), \mathbb{O}^{-1}(k) \rangle \in G\,, \\ 0, & \textit{otherwise}\,. \end{cases}$$

Contrary to adjacency matrices, a tensorial representation allows a simple solution for handling multiple edges between two nodes. Given a RDF tensor, we observe that the majority of its elements will be zero, *i.e.*, the originating graph is loosely connected [1]. It is therefore advisable to employ a *rule notation* to express a tensor, instead of listing all its elements. With the rule notation, we will express a tensor with a list of triples $\{i, j, k\} \to r_{ijk}$, for all $r_{ijk} \neq 0$, and assuming all other elements being zero, if not present in the triples list.

EXAMPLE 1. *Let us consider the RDF graph in Figure 2. The* RDF tensor $\mathcal{R}$ *of G can be therefore given as shown in Figure 3. In the Figure, for typographical simplicity, we omitted*

$$0_{\mathcal{R}} = (0, 0, 0, 0, 0, 0, 0)^t$$

denoted with a dash. A more concise way of expressing the above tensors is by employing the rule notation, i.e.*, assuming zero as the default value, and listing all non-zero elements:*

$$\mathcal{R} = \{\,\{1, 3, 1\} \to 1\,,\ \{1, 4, 3\} \to 1\,,\ \ldots, \{3, 1, 13\} \to 1\,\}\ .$$

For instance the element $\{1, 3, 1\} \to 1$ means that there exists in G the triple $\langle \mathbb{S}^{-1}(1), \mathbb{P}^{-1}(3), \mathbb{O}^{-1}(1) \rangle$, i.e.*, $\langle a, hates, b \rangle$.*

SPARQL. Abstractly speaking, a SPARQL query Q mainly is a 5-tuple of the form $\langle qt, RC, DD, G_P, SM \rangle$, where: qt is the *query type*, RC is the *result clause*, DD is the *dataset definition*, G_P is the *graph pattern*, and SM is the *solution modifier*. At the heart of Q there lies the *graph pattern* G_P that searches for specific subgraphs in the input RDF dataset. Its result is a (multi) set of *mappings*, each of which associates variables to elements of $\mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$. In particular the official SPARQL syntax considers operators UNION, OPTIONAL, FILTER and *concatenation* via a point symbol " . " to construct graph patterns.

DEFINITION 5 (GRAPH PATTERN). *A graph pattern G_P is a 4-tuple $\langle \mathbb{T}, f, OPT, U \rangle$ where*

- *$\mathbb{T}$ is a set of* triple patterns *$\{t_1, \ldots, t_n\}$ that may contain a* variable*,* i.e.*, a symbolic name starting with a ? and can match any node (resource or literal) in the RDF dataset;*
- *f is a* FILTER *constraint using boolean conditions to filter out unwanted query results;*
- *OPT is a set of* OPTIONAL *statements trying to match $\mathbb{T}$, but the whole query does not fail if the optional statements do not match. This set is modeled as G_P;*
- *U is a set of* UNION *statements modeled as G_P.*

The *result clause* identifies which information to return from the query. It returns a table of variables (occurring in G_P) and values that satisfy the query. The *dataset definition* is optional and specifies the input RDF dataset to use during pattern matching. If it is absent, the query processor itself determines the dataset to use. The optional *solution-modifier* allows sorting of the mappings obtained from the pattern matching, as well as returning only a specific window of mappings (*e.g.*, mappings 1 to 10). The result is a list L of mappings. The output of the SPARQL query is then determined by the *query-type*: SELECT, ASK, CONSTRUCT and DESCRIBE.

EXAMPLE 2. *Let us consider the RDF graph shown in Figure 2 and three different SPARQL queries over such graph (*i.e.*, we used simple terms in place of verbose* URI*s).*

```
Q1: SELECT ?x ?y1
    WHERE { ?x type Person. ?x hobby 'CAR'.
            ?x name ?y1. ?x mbox ?y2. ?x age ?z.
             FILTER (xsd:integer(?z) >= 20) }
Q2: SELECT *
    WHERE { {?x name ?y} UNION {?z mbox ?w} }
Q3: SELECT ?z ?y ?w
    WHERE { ?x type Person. ?x friendOf ?y. ?x name ?z.
            OPTIONAL { ?x mbox ?w. } }
```

Q1 *selects* URI *and name of persons having the hobby of cars, a name, a mailbox and an age greater (or equal) than twenty.* Q2 *selects* URI *and name of persons united to* URI *and mailbox of persons. Finally* Q3 *selects the name and (in case) the mailbox of all persons having a friend (of which the query returns the* URI *also).*

Referring to Example 2, as illustrated above (in particular refer to definition 5), for instance we model *Q1* as $\langle$SELECT, $\{?x\}, \_, G_P, \_\rangle$ with $G_P = \langle \mathbb{T}, f, \_, \_\rangle$, $\mathbb{T} = \{\langle ?x, type, Person\rangle, \langle ?x, hobby, Car\rangle, \langle ?x, name, ?y1\rangle, \langle ?x, mbox, ?y2\rangle, \langle ?x, age, ?z\rangle\}$ and $f = \{?z >= 20\}$.

In [21], the authors analyzed a log of SPARQL queries harvested from the DBPedia SPARQL Endpoint from April to July 2010. The log analysis produced interesting statistics, allowing us to simplify the features of a SPARQL query. In the following we will consider a query Q as a 2-tuple of the form $\langle RC, G_P \rangle$, *i.e.* only SELECT queries with *result clause* and *graph pattern*, employing the operators {AND, FILTER, OPTIONAL, UNION}. This simplification does not compromise the feasibility and generality of the approach.

3. SPARQL DOF MODELING

This section is devoted to the introduction of our approach to SPARQL selection query treatment. In the rest of the paper we will use the term "triple" to indicate also a triple pattern (*i.e.*, a triple can be considered a triple pattern $\langle s, p, o \rangle$, where s, p and o are constants).

$$\begin{pmatrix} \begin{pmatrix}0\\0\\1\\0\\0\\0\\0\end{pmatrix} & - & \begin{pmatrix}0\\0\\0\\1\\0\\0\\0\end{pmatrix} & - & - & - & - & \begin{pmatrix}0\\0\\0\\0\\0\\1\\0\end{pmatrix} & \begin{pmatrix}0\\0\\0\\0\\0\\0\\1\end{pmatrix} & \begin{pmatrix}0\\0\\0\\0\\1\\0\\0\end{pmatrix} & \begin{pmatrix}1\\0\\0\\0\\0\\0\\0\end{pmatrix} & - & - \\ - & \begin{pmatrix}0\\1\\0\\0\\0\\0\\0\end{pmatrix} & - & \begin{pmatrix}0\\0\\0\\0\\0\\1\\0\end{pmatrix} & - & - & - & - & \begin{pmatrix}0\\0\\0\\0\\0\\0\\1\end{pmatrix} & - & - & \begin{pmatrix}1\\0\\0\\0\\0\\0\\0\end{pmatrix} & - \\ \begin{pmatrix}0\\1\\0\\0\\0\\0\\0\end{pmatrix} & - & \begin{pmatrix}0\\0\\0\\1\\0\\0\\0\end{pmatrix} & - & \begin{pmatrix}0\\0\\0\\0\\0\\1\\0\end{pmatrix} & \begin{pmatrix}0\\0\\0\\0\\1\\0\\0\end{pmatrix} & \begin{pmatrix}0\\0\\0\\0\\1\\0\\0\end{pmatrix} & - & \begin{pmatrix}0\\0\\0\\0\\0\\0\\1\end{pmatrix} & - & - & - & \begin{pmatrix}1\\0\\0\\0\\0\\0\\0\end{pmatrix} \end{pmatrix}.$$

$\mathbb{S}$	S
1	a
2	b
3	c

$\mathbb{P}$	P
1	age
2	$friendOf$
3	$hobby$
4	$hates$
5	$mbox$
6	$name$
7	$type$

$\mathbb{O}$	O
1	b
2	c
3	Car
4	$John$
5	$Mary$
6	$m1@ex.com$
7	$m2@ex.it$
8	$p@ex.it$
9	$Paul$
10	$Person$
11	18
12	23
13	28

Figure 3: An example of RDF tensor $\mathcal{R}$ and corresponding RDF Sets Indexing $\mathbb{S}$, $\mathbb{P}$ and $\mathbb{O}$.

3.1 Triple Analysis

A selection query consists of a SELECT clause, followed by a list of *unbounded* variables, *i.e.*, variables whose value is not calculated yet. A WHERE clause states all the conditions the variables must meet, conjunctively.

DEFINITION 6 (DEGREE OF FREEDOM). *The* degree of freedom *of a triple t, $dof : \mathbb{T} \longrightarrow \{+3, +1, -1, -3\}$, is the function defined as: $dof(t) := v - k$, being k and v the number of constants and variables in t, respectively.*

The *degree of freedom*, or DOF, is a measure of a triple's explicit constraints, hence a condition with no constraints (*i.e.*, constituted by variables) has the highest DOF, while one constituted by only constants has the lowest DOF.

EXAMPLE 3. *Referring to Figure 2, the triple $t_1 := \langle a, hates, b\rangle$ is constituted by three constants, and no variables: its DOF is $dof(t_1) = v - k = 0 - 3 = -3$. A triple as $t_2 := \langle a, hates, ?x\rangle$ has two constants, and one variable, namely ?x. Its degree of freedom is $dof(t_2) = v-k = 1-2 = -1$. With $t_3 := \langle ?x, hates, ?y\rangle$ we express a constraint constituted by one constant, hates, and two variables. It follows that $dof(t_3) = v - k = 2 - 1 = +1$. Last, let $t_4 := \langle ?x, ?y, ?z\rangle$ be a triple: it is composed by three variables, and no constants. Therefore, $dof(t_4) = v - k = 3 - 0$=+3.*

3.2 Constraint Solving

Due to the fact that triples may have various degrees of freedom, *i.e.*, they may or may not be bound to constants, in the following we shall consider each case, and calculate the results of a triple within our tensorial framework. In particular we employ the vector specification of Kroneker tensor, commonly known as *Kroneker delta* (δ). For instance, given the RDF tensor $\mathcal{R}_{ijk}$, the notation δ_i^2 means that each component in a position different from 2 has value 0, while the component in position 2 has value 1. The number of components in δ_i^2 is equal to the size of the dimension i in $\mathcal{R}_{ijk}$. Referring to the RDF tensor of Figure 3, $\delta_i^2 = (0, 1, 0)$. For the sake of conciseness, we employ a simplified *Einstein's summation convention*: if two adjoined entities share a common index, a summation is implicitly intended, *e.g.*, $\mathcal{R}_{ijk}\,\delta_i^2 := \sum_i \mathcal{R}_{ijk}\,\delta_i^2$.

Degree −3. A triple t with $dof(t) = -3$, is by definition bound to three constants $c_1 \in \mathcal{S}$, $c_2 \in \mathcal{P}$, and $c_3 \in \mathcal{O}$, and therefore the constraint is computed as $\mathcal{R}_{ijk}\,\delta_i^{\mathbb{S}(c_1)}\,\delta_j^{\mathbb{P}(c_2)}\,\delta_k^{\mathbb{O}(c_3)}$.

Degree −1. A triple t with $dof(t) = -1$ possesses two constant values, c_1, and c_2, whose domains are associated with their respective indices i_1 and i_2. The results are hence given by $\mathcal{R}_{ijk}\,\delta_{i_1}^{\mathbb{K}_1(c_1)}\,\delta_{i_2}^{\mathbb{K}_2(c_2)}$ with $\mathbb{K}_1, \mathbb{K}_2 \in \{\mathbb{S}, \mathbb{P}, \mathbb{O}\}$. The result of such computation is a *vector* bound the only variable present in the triple, and, with the rule notation, it may take the form of a list of values.

Degree +1. A triple t with $dof(t) = +1$ present two variables, and a single constant c, with the index i_c associated to its domain. So, we may compute the constrains as $\mathcal{R}_{ijk}\,\delta_{i_c}^{\mathbb{K}(c)}$ with $\mathbb{K} \in \{\mathbb{S}, \mathbb{P}, \mathbb{O}\}$. The above equation yields a rank-2 tensor, or in other words, a *matrix*, and promptly interpreted as a list of *couples* when employing the rule notation.

Degree +3. A triple t with $dof(t) = +3$ is associated to no constants, and therefore its result is unbounded, *i.e.*, it is computed by returning $\mathcal{R}_{ijk}$.

3.3 Conjunctive Operations

A list of triples in the set $\mathbb{T}$ of a given SELECT query must be satisfied *conjunctively*.

DEFINITION 7 (DISJOINED TRIPLES). *Let $t_1, t_2 \in \mathbb{T}$ be two triples; t_1 and t_2 are* disjoined *if they share no common variables.*

With the above definition, we are able now to focus on the only two cases that may present in a selective SPARQL query.

Disjoined Triples. Given two disjoined triples, their conjunction is simply the union of their bounded variables. By definition, if a variable is bound to an *empty set*, the query yields no results.

Conjoined Triples. Sharing at least one variable, two conjoined constraints t_1 and t_2 produce their results by applying the *Hadamard product* (element-wise multiplication denoted by $\circ$) on the common variables: being $u = (u_i)$ and $v = (v_i)$ we have $z = (z_i) = u \circ v := (u_i \cdot v_i)$.

EXAMPLE 4. *With reference to Figure 2, let t_1 and t_2 be two* conjoined triples, *with $t_1 := \langle ?x, friendOf, c\rangle$, and $t_2 := \langle a, hates, ?x\rangle$. The first triple will be computed as $t_1 := \mathcal{R}_{ijk}\,\delta_j^{\mathbb{P}(friendOf)}\,\delta_k^{\mathbb{O}(c)}$,*

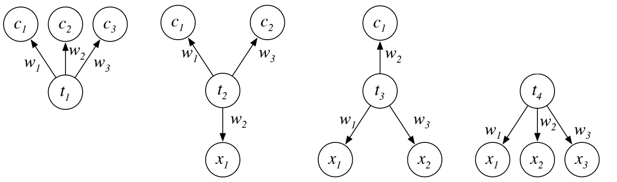


Figure 4: Triples by their *degree of freedom* in an *execution graph*. From left to right, we have DOFs -3, -1, $+1$, and $+3$.

which yields the vector $t_1 := \{\{\mathbb{S}(b)\} \rightarrow 1\}$. Analogously, t_2 will give the result $t_2 := \mathcal{R}_{ijk}\, \delta_i^{\mathbb{S}(a)} \delta_j^{\mathbb{P}(hates)} = \{\{\mathbb{O}(b)\} \rightarrow 1\}$. Hence, $t_1 \circ t_2 = \{\{\mathbb{S}(a)\} \rightarrow 1\}$ is the final result, i.e. t_1 and t_2 have b *in common. Conversely, if we have $t_2 := \langle a, friendOf, ?x\rangle$, this will yield no results, since $t_2 := \varnothing$, and therefore $t_1 \circ t_2 = \varnothing$.*

Anticipating an implementation aspect, we highlight the fact that *conjoined triples* may be computed sequentially for each value in a variable. In other words, if we have a shared variable $?x$ to which there is associated the set of values $\{v_1, v_2\}$, the query shall be processed for v_1, and separately for v_2.

4. QUERY ANSWERING

Given a query, we may now proceed in describing a suitable scheduling algorithm for the analysis of all triples.

4.1 Query Scheduling

Let us first recognize that the final purpose of a scheduling is to determine the *order of execution* of each triple. In turn, this will bind all the variables in the query to their respective values, if any. The degree of freedom of each triple is the primary tool for determining a sequence in which constraints should be solved: it indicates the priority of each triple, with lowest DOFs associated to higher priorities. A *directed acyclic graph* (DAG) may represent an algorithm [3], and in our case, will be employed to visually select all triples for execution.

DEFINITION 8 (EXECUTION GRAPH). *Given a set $\mathbb{T}$ of triples, an* execution graph *on $\mathbb{T}$ is a weighted directed acyclic graph $EG = (N, E)$. N is the set of nodes resulting from $N_t \cup N_c \cup N_v$, where N_t is the set of triples in $\mathbb{T}$, N_c and N_v are the sets of* constants *and* variables *associated to the triples of $\mathbb{T}$, respectively. E is the set of weighted edges connecting triples to their respective constants and variables, with weights representing the domain (*i.e.*, $\mathcal{S}$, $\mathcal{P}$ or $\mathcal{O}$) of the ending node.*

In order to enhance readability, we present the execution graph in a three-layered fashion, as depicted in Figure 4. The center layer contains all triples N_t, the top layer the constants N_c, and the bottom layer the variables N_v.

EXAMPLE 5. *With reference to the query* Q1 *of Example 2 and Figure 4, we rewrite* Q1 *as follows. The triple $t_1 := \langle ?x, type, Person\rangle$ has DOF -1 and is represented by the second graph of Figure 4, with $c_1 = type$, $c_2 = Person$, and their respective weights are $w_1 = \mathcal{P}$, $w_2 = \mathcal{S}$, and $w_3 = \mathcal{O}$. Hence, the computation of t_1 is given by $\mathcal{R}_{ijk}\ \delta_j^{\mathbb{P}(type)}\ \delta_k^{\mathbb{O}(Person)}$, giving birth to a vector of elements, bound to the variable $?x$. The triple $t_2 := \langle ?x, hobby, car\rangle$ has DOF -1 and is represented similarly to t_1. Therefore we may compute t_2 as $\mathcal{R}_{ijk}\ \delta_j^{\mathbb{P}(hobby)}\ \delta_k^{\mathbb{O}(car)}$. The triples $t_3 := \langle ?x, name, ?y1\rangle$, $t_4 := \langle ?x, mbox, ?y2\rangle$, $t_5 := \langle ?x, age, ?z\rangle$ have DOF $+1$ and are represented by the third graph of Figure 4. For instance, referring to t_3, we have only one constant $c_1 = name$ with weight $w_2 = \mathcal{P}$, and two variables $x_1 = ?x$ and $x_2 = ?y1$; their respective weights are $w_1 = \mathcal{S}$, $w_3 = \mathcal{O}$. Hence, t_3 is computed as $\mathcal{R}_{ijk}\ \delta_j^{\mathbb{P}(name)}$: the matrix resulting from the computation is the set of couples associated to the variables $?x$ and $?y1$. Figure 5 shows the final execution graph built from* Q1.

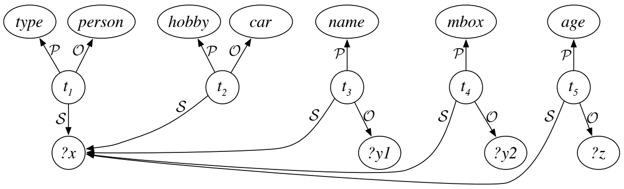


Figure 5: An example of execution graph.

The scheduling for the execution of a SELECT SPARQL query is therefore dynamically determined. In our framework, given a set $\mathbb{T}$ of triples, the scheduling algorithm can be sketched as follows:

1. Determine the DOF of each triple $t_i \in \mathbb{T}$;
2. Select the triple $\widetilde{t} \in \mathbb{T}$ with lowest DOF;
3. Execute $\widetilde{t}$ as described in Section 3.2;
4. Bind all variables in the triples of $\mathbb{T}$ conjunctively;
5. Remove $\widetilde{t}$ from the list;
6. If $\mathbb{T} = \varnothing$ *stop*; else proceed to *step* 1.

In the previous scheduling schema, we may encounter triples with the same DOF. In this case, in Step 2 we shall select the triple which raises the DOF of the largest numer of triples in a query, excluding itself. Suppose for instance that our triple patterns are as follows: *?x name ?y*, *?x hobby ?u*, *?u color ?z*, and finally the pattern *?u model ?w*. In the example we may notice as every triple has DOF equal to $+1$. However, analyzing the prospect DOFs, we notice as the first will promote only the second query through *?u*, both the third and fourth will affect two patterns—the second and fourth—while the second will affect all queries, and hence it is selected for processing.

EXAMPLE 6. *This example will elucidate the process of query answering via DOF analysis. With reference to the RDF tensor $\mathcal{R}$ described in Figure 3, let the SPARQL query under scrutiny be Q1 = ?x type Person. ?x hobby 'CAR'. ?x name ?y1. ?x mbox ?y2. ?x age ?z.; as discussed above, we have five triples t_1, t_2, t_3, t_4, and t_5. Given our five constraints we hence proceed in analyzing their* degrees of freedom, *resulting in $dof(t_1) = dof(t_2) = 1$, and $dof(t_3) = dof(t_4) = dof(t_5) = +1$. We may proceed now in executing the query. First, let us remind that any variable is currently* unbounded, *or in other words, unassociated to any value. The first triple to be computed is between $t_1 := \langle ?x, type, Person\rangle$ and $t_2 := \langle ?x, hobby, car\rangle$, having the lowest value of DOF, equal to -1. In this case we start from t_1. Therefore we determine the values associated to this constraint. This triple produces a* vector *of values to be bound to $?x$:*

$$\begin{aligned} t_1 \ &:= \mathcal{R}_{ijk}\ \delta_j^{\mathbb{P}(type)}\ \delta_k^{\mathbb{O}(person)} = \\ &= \{\,\{\mathbb{S}(a)\} \rightarrow 1, \{\mathbb{S}(b)\} \rightarrow 1, \{\mathbb{S}(c)\} \rightarrow 1\,\}\ . \end{aligned}$$

This computation returns the set X of values $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ to be associated to the variable $?x$. Therefore we have to reanalyze the *degree of freedom of the remaining triples. In this case we have $dof(t_2) = -3$ and $dof(t_3) = dof(t_4) = dof(t_5) = -1$,* i.e., *the variable $?x$ is promoted to the role of constant. The next triple to be computed is t_2, having the highest value of DOF, equal to -3. Therefore, for each $x_z \in X$, the triple yields the boolean value*

$$t_2 := \mathcal{R}_{ijk}\ \delta_i^{\mathbb{S}(x_z)}\ \delta_j^{\mathbb{P}(hobby)}\ \delta_k^{\mathbb{O}(car)} = 1\,.$$

Since the outcome of the computation is true *for $x_z \in \{a, c\}$, the set X is filtered accordingly, and the query processing may proceed. Proceeding our scheduling, t_3 has to be processed similarly to t_1. This triple produces a* vector *of values to be bound to $?y1$:*

$$\begin{aligned} t_3 \quad &:= \mathcal{R}_{ijk}\ \delta_i^{\mathbb{S}(a)}\ \delta_j^{\mathbb{P}(name)} \cup \mathcal{R}_{ijk}\ \delta_i^{\mathbb{S}(c)}\ \delta_j^{\mathbb{P}(name)} = \\ &= \{\,\{\mathbb{O}(Paul)\} \to 1, \{\mathbb{O}(Mary)\} \to 1\,\}\ . \end{aligned}$$

Owing to the non-emptiness of the previous computation, we proceed to the last triples t_4 and then t_5 computed as t_3 producing the vectors of values to be bound to $?y2$ and $?z$, respectively.

$$\begin{aligned} t_4 := \quad & \{\,\{\mathbb{O}(p@ex.it)\} \to 1, \{\mathbb{O}(m1@ex.it)\} \to 1, \\ & \{\mathbb{O}(m2@ex.com)\} \to 1\,\}\,. \\ t_5 := \quad & \{\,\{\mathbb{O}(18)\} \to 1, \{\mathbb{O}(28)\} \to 1\,\}\ . \end{aligned}$$

In both the triples all values in the set X, i.e., *a and c, provide non-empty results in the computation. Finally we apply the filter to the values associated to the variable $?z$,* i.e., *$?z \geq 20$. Consequently, we have to filter t_5 to $\{\{\mathbb{O}(28)\} \to 1\}$ and then the set X,* i.e., *$X = \{c\}$. Since all triples were processed, the scheduling stops. Moreover we bind the set $Y1$ of values associated to $?y1$ to X obtaining $\{Mary\}$. Because the result clause of Q1 is $?x$ $?y1$, we return the so-generated X and $Y1$.*

In the rest of this section we provide an implementation of our scheduling algorithm to perform both "conjunctive" and "non-conjunctive" SELECT SPARQL queries.

4.2 Conjunctive Pattern with Filters

The so-called conjunctive pattern with filters (CPF) uses only the operators AND and FILTER. Therefore our scheduling algorithm takes as input a set $\mathbb{T}$ of triple patterns $t_1, \ldots, t_m$, a filter f, a set X_v of result clause variables $?x_1, \ldots, ?x_n$ and a RDF tensor $\mathcal{R}$, in terms of sum of p chunks $\mathcal{R}_i$; p is the number of processes on p hosts, while $\mathcal{R}_i$ is the slice of $\mathcal{R}$ corresponding to the set of triples in the i-th host. The output is a set $\mathcal{X}_I$ of instances $X_1, \ldots, X_n$, that is, each X_i contains values to be associated to the variable $?x_i$ such that all constraints $t_1, \ldots, t_m$ are satisfied. The set $\mathcal{X}_I$ is computed as shown in Algorithm 1.

More in detail, we initialize a map V where the keys are all the variables occurring in the triples of $\mathbb{T}$ while to each key we associate a set of values, i.e. at the beginning an empty set (lines [1-2]). The procedure getVariables is responsible to extract all variables from the triple patterns in $\mathbb{T}$. For instance referring to the query Q1 of Example 2 on the RDF graph of Figure 2, we have $\mathbb{T} = \{t_1, t_2, t_3, t_4, t_5\}$ as described above, $f = ?z \geq 20$, $X_v = \{?x, ?y1\}$ and the RDF tensor $\mathcal{R}$ illustrated in Figure 3. The map V is initialized to $\{\langle ?x, \varnothing\rangle, \langle ?y1, \varnothing\rangle, \langle ?y2, \varnothing\rangle, \langle ?z, \varnothing\rangle\}$.

Algorithm 1: Execution of a SPARQL query

```
Input  : $\mathbb{T} = \{t_1, \ldots, t_m\}$, $f$, $X_v = \{?x_1, \ldots, ?x_n\}$,
         $\mathcal{R} = \mathcal{R}_1 + \ldots + \mathcal{R}_p$
Output: $\mathcal{X}_I = \{X_1, \ldots, X_n\}$

1  V ← ∅;
2  foreach $?x \in$ getVariables($\mathbb{T}$) do V.put($?x$,∅);
3  proceed ← true;
4  while ($\mathbb{T}$ is not empty) ∧ (proceed) do
5      $t \leftarrow \mathbb{T}$.dequeue();
6      broadcast($t$);
7      proceed ← reduce(Application($t$, V, $\mathcal{R}_i$), OR);
8      if (proceed) then
9          Update($\mathbb{T}$, V);
10         Filter(V,$f$);
11         foreach $?x \in$ getVariables($\{t\}$) do
12             reduce(V.get($?x$), sum);
13 if (proceed) then
14     foreach $?x \in X_v$ do $\mathcal{X}_I \leftarrow \mathcal{X}_I \cup$ V.get($?x$);
15 else $\mathcal{X}_I \leftarrow \varnothing$;
16 return $\mathcal{X}_I$;
```

We organize $\mathbb{T}$ as a priority queue (i.e. high priority corresponds to low DOF associated to a constraint t). Then we extract a constraint t from $\mathbb{T}$ until $\mathbb{T}$ is not empty and the computation of t produces a non-empty result (lines [4-12]). In particular, a broadcast mechanism sends t and V to all hosts, which compute t on the own $\mathcal{R}_\rangle$. The resulting values associated to the variables occurring in t are included in the set V and then filtered by applying f (*i.e.*, the procedure Filter is responsible of such task) in terms of a map operation: being $u = (u_i)$ and f a suitable function, $v = (v_i) = \mathrm{map}(f, u) := (\,f(u_i)\,)$. Consequently, we update the DOFs of each triple in $\mathbb{T}$ (*i.e.*, the procedure Update is responsible of such task). At the end, if all triples brought result we return the sets of values associated to the variables in X_v, otherwise we return an empty set.

Algorithm 2: Tensor application of a triple

```
Input  : $t$, V = $\{\langle ?x_1, X_1\rangle, \ldots, \langle ?x_z, X_z\rangle\}$, $\mathcal{R}$
Output: boolean

1  if isVariable($t.s$) then $S \leftarrow$ V.get($t.s$);
2  else $S \leftarrow S \cup \{t.s\}$;
3  if isVariable($t.p$) then $P \leftarrow$ V.get($t.p$);
4  else $P \leftarrow P \cup \{t.p\}$;
5  if isVariable($t.o$) then $O \leftarrow$ V.get($t.o$);
6  else $O \leftarrow O \cup \{t.o\}$;
7  switch dof($t$,V) do
8      case -3
9          return CASETHREE($t$, $S$, $P$, $O$, V, $\mathcal{R}$);
10     case -1
11         return CASEONE($t$, $S$, $P$, $O$, V, $\mathcal{R}$);
12     case +1
13         return CASEMINUSONE($t$, $S$, $P$, $O$, V, $\mathcal{R}$);
14     case +3
15         V.put($t.s$, getValues($\mathcal{R}_{ijk}\bar{1}_j\bar{1}_k$));
16         V.put($t.p$, getValues($\mathcal{R}_{ijk}\bar{1}_i\bar{1}_k$));
17         V.put($t.o$, getValues($\mathcal{R}_{ijk}\bar{1}_i\bar{1}_j$));
18         return true;
19     otherwise
20         return false;
```

The computation of a triple t is performed by the procedure Application. Since we are in a distributed environment, we exploit a reduce function that takes as input a set of values and an operator to combine such values. Since Application returns a boolean value (*i.e.* true if the tensor application was able to compute t), the reduce function takes all boolean values from each host and combine them through the OR logic operator (line [7]). Similarly, if the result of such reduce is true, then we combine all the values retrieved for each variable $?x$ in t by the hosts by reducing them though a sum operator, *i.e.* union, (lines [11-12]).As shown in Algorithm 2, it takes as input the triple t, the map V and the tensor slice $\mathcal{R}$. In this procedure we have to evaluate the DOF of t: how many constants (or variables to which there exists a non-empty set associated in V) and how many variables (to which an empty set is associated in V). We use the notation $t.s$, $t.p$ and $t.o$ to access to subject, property and object of a triple t, respectively. The procedure `isVariable` evaluates if a component of t is a variable: we extract all values associated in the previous computations and we put them in the sets S, P and O. Otherwise S, P and O contain the constants $t.s$, $t.p$ and $t.o$, respectively. W.r.t the DOF, we call the corresponding procedure implementing the tensor application on $\mathcal{R}$.

Case -3. The triple t has DOF -3; in this case we have to filter all $s \in S$, $p \in P$ and $o \in O$ such that there does not exist a triple $\langle s, p, o \rangle$ in $\mathcal{R}$. As illustrated in Algorithm 3, we iterate on S, P and O to compute the set D of elements to filter. If all S, P and O are not empty we can proceed with our scheduling.

Algorithm 3: Case with DOF -3

```
Input  : $t$, $S$, $P$, $O$, V, $\mathcal{R}$
Output: boolean
$D \leftarrow \varnothing$;
foreach $s \in S$ do
    foreach $p \in P$ do
        foreach $o \in O$ do
            if $\mathcal{R}_{\mathbb{S}(s)\mathbb{P}(p)\mathbb{O}(o)} = 0$ then
                $D \leftarrow D \cup \{s\}$;
                $D \leftarrow D \cup \{p\}$;
                $D \leftarrow D \cup \{o\}$;
            else
                $D \leftarrow D - \{s\}$;
                $D \leftarrow D - \{p\}$;
                $D \leftarrow D - \{o\}$;
$S \leftarrow S - D$;
$P \leftarrow P - D$;
$O \leftarrow O - D$;
if isVariable($t.s$) then V.put($t.s$,$S$);
if isVariable($t.p$) then V.put($t.p$,$P$);
if isVariable($t.o$) then V.put($t.o$,$O$);
return ($S \neq \varnothing \wedge P \neq \varnothing \wedge O \neq \varnothing$);
```

Case -1. The triple t has DOF -1; in this case t provides only one variable. As shown in Algorithm 4, the procedure `roleVariable` evaluates which component of t is a variable (*i.e.*, `'s'` for subject, `'p'` for property, `'o'` for object). We have to compute the application $\mathcal{R}_{ijk}\ \delta_{i_1}^{\mathbb{K}_1(c_1)}\ \delta_{i_2}^{\mathbb{K}_2(c_2)}$ on the two constants c_1 and c_2 of t. For instance if `roleVariable`(t) is `'s'` then we employ the constants $p \in P$ and $o \in O$ (*i.e.*, coming from the previous computations) and we compute $\mathcal{R}_{ijk}\ \delta_j^{\mathbb{P}(p)}\ \delta_k^{\mathbb{O}(o)}$. The procedure `getValues` retrieves the values associated to the resulting vector and put them in the set X. Finally we associate X to the variable $t.s$ in V. If the resulting set X is not empty we can proceed with our scheduling.

Case $+1$. The triple t has DOF $+1$; in this case t provides only one constant. As shown in Algorithm 5, the procedure `roleConstant` evaluates which component of t is a constant.

We have to compute $\mathcal{R}_{ijk}\ \delta_{i_c}^{\mathbb{K}(c)}$. For instance if `roleConstant`(t) is `'s'`, we have to extract all predicates and objects associated to the elements $e \in S$ (coming from the previous computations in case). Therefore we compute $\mathcal{R}_{\mathbb{S}(e)jk}\bar{1}_k$ and $\mathcal{R}_{\mathbb{S}(e)jk}\bar{1}_j$; $\mathcal{R}_{\mathbb{S}(e)jk}$ returns a matrix fixing the dimension i to $\mathbb{S}(e)$ while $\bar{1}_j$ ($\bar{1}_k$) is a vector with all components 1 and with length equal to the size of the dimension j (k). All the properties (E_1) and objects (E_2) are then associated to $t.p$ and $t.o$ in the map V. If E_1 and E_2 are non empty our scheduling can proceed.

Algorithm 4: Case with DOF -1

```
Input  : $t$, $S$, $P$, $O$, V, $\mathcal{R}$
Output: boolean
$X \leftarrow \varnothing$;
switch roleVariable($t$) do
    case 's'
        foreach $p \in P$ do
            foreach $o \in O$ do
                $X \leftarrow X \cup$ getValues($\mathcal{R}_{ijk}\delta_j^{\mathbb{P}(p)}\delta_k^{\mathbb{O}(o)}$);
        V.put($t.s$,$X$);
    case 'p'
        foreach $s \in S$ do
            foreach $o \in O$ do
                $X \leftarrow X \cup$ getValues($\mathcal{R}_{ijk}\delta_i^{\mathbb{S}(s)}\delta_k^{\mathbb{O}(o)}$);
        V.put($t.p$,$X$);
    case 'o'
        foreach $s \in S$ do
            foreach $p \in P$ do
                $X \leftarrow X \cup$ getValues($\mathcal{R}_{ijk}\delta_i^{\mathbb{S}(s)}\delta_j^{\mathbb{P}(p)}$);
        V.put($t.o$,$X$);
    otherwise
        return false;
return $X \neq \varnothing$;
```

Case $+3$. If the triple t has DOF $+3$ we have to extract all subjects ($\mathcal{R}_{ijk}\bar{1}_j\bar{1}_k$), properties ($\mathcal{R}_{ijk}\bar{1}_i\bar{1}_k$) and objects ($\mathcal{R}_{ijk}\bar{1}_i\bar{1}_j$) from $\mathcal{R}$.

4.3 Non-Conjunctive Pattern with Filters

The so called non conjunctive pattern with filters (non-CPF) employs OPTIONAL and UNION, beyond AND and FILTER. In this case our scheduling algorithm has to perform disjoined triples.

Union. Given a query $\langle RC, G_P \rangle$ with $G_P = \langle \mathbb{T}, f, \_, U \rangle$ (*i.e.*, $U = \langle \mathbb{T}_U, f_U, \_, \_ \rangle$), we perform our scheduling algorithm on the triples of both $\mathbb{T}$ and $\mathbb{T}_U$, separately. Finally we make the union of all $\mathcal{X}_I$. For instance let us consider the query Q2 of Example 2. We have $\mathbb{T} = \{t_1\}$ and $\mathbb{T}_U = \{t_2\}$, where $t_1 := \langle ?x, name, ?y \rangle$ and $t_2 := \langle ?z, mbox, ?w \rangle$. From $\mathbb{T}$ we generate $\mathcal{X}_I = \{\{a, b, c\}, \{Paul, John, Mary\}\}$ while from $\mathbb{T}_U$ we have $\mathcal{X}_I = \{\{a, c\}, \{p@ex.it, m1@ex.it, m2@ex.com\}\}$.

The final result is $\mathcal{X}_I = \{\ \{a, b, c\},\ \{Paul, John, Mary\},$
$\{\ p@ex.it,\ m1@ex.it,$
$m2@ex.com\}\ \}$.

Optional. Given a query $\langle RC, G_P \rangle$ with $G_P = \langle \mathbb{T}, f, OPT, \_ \rangle$ (*i.e.*, $OPT = \langle \mathbb{T}_{OPT}, f_{OPT}, \_, \_ \rangle$), we perform our scheduling algorithm on the triples of both $\mathbb{T}$ and $\mathbb{T} \cup \mathbb{T}_{OPT}$, separately. Finally we make the union of all $\mathcal{X}_I$. For instance let us consider the query Q3 of Example 2, we have $\mathbb{T} = \{t_1, t_2, t_3\}$ and $\mathbb{T}_{OPT} = \{t_4\}$, where $t_1 := \langle ?x, type, Person \rangle$, $t_2 := \langle ?x, friendOf, ?y \rangle$, $t_3 := \langle ?x, name, ?z \rangle$ and $t_4 := \langle ?x, mbox, ?w \rangle$. From $\mathbb{T}$ we generate $\mathcal{X}_I = \{John, Mary\}, \{b, c\}\}$, while from $\mathbb{T} \cup \mathbb{T}_{OPT}$ we have $\mathcal{X}_I = \{\{Mary\}, \{b\}, \{m1@ex.it,\ m2@ex.com\}\}$. The final result is $\mathcal{X}_I = \{\{John, Mary\}, \{b, c\}, \{m1@ex.it,\ m2@ex.com\}\}$.

Of course, both the graph patterns U and OPT can be more complex, i.e. with other UNION or OPTIONAL statements; in this

Algorithm 5: Case with DOF +1

```
Input  : $t$, $S$, $P$, $O$, V, $\mathcal{R}$
Output: boolean
$E_1 \leftarrow \varnothing$;
$E_2 \leftarrow \varnothing$;
switch roleConstant ($t$) do
    case 's'
        foreach $e \in S$ do
            $E_1 \leftarrow E_1 \cup$ getValues ($\mathcal{R}_{\mathbb{S}(e)jk}\bar{1}_k$);
            $E_2 \leftarrow E_2 \cup$ getValues ($\mathcal{R}_{\mathbb{S}(e)jk}\bar{1}_j$);
        V.put ($t.p$, $E_1$);
        V.put ($t.o$, $E_2$);
    case 'p'
        foreach $e \in P$ do
            $E_1 \leftarrow E_1 \cup$ getValues ($\mathcal{R}_{i\,\mathbb{P}(e)k}\bar{1}_k$);
            $E_2 \leftarrow E_2 \cup$ getValues ($\mathcal{R}_{i\,\mathbb{P}(e)k}\bar{1}_i$);
        V.put ($t.s$, $E_1$);
        V.put ($t.o$, $E_2$);
    case 'o'
        foreach $e \in O$ do
            $E_1 \leftarrow E_1 \cup$ getValues ($\mathcal{R}_{ij\,\mathbb{O}(e)}\bar{1}_j$);
            $E_2 \leftarrow E_2 \cup$ getValues ($\mathcal{R}_{ij\,\mathbb{O}(e)}\bar{1}_i$);
        V.put ($t.s$, $E_1$);
        V.put ($t.p$, $E_2$);
    otherwise
        return false;
return ($E_1 \neq \varnothing \wedge E_2 \neq \varnothing$);
```

case we apply the above procedure recursively. Concluding, once our scheduling algorithm produced $\mathcal{X}_I$, we demand to a *front-end* task the presentation of results in terms of tuples, conforming to the result clause of the query.

5. IMPLEMENTATION

Our prime objective is to provide a general storage, in memory data structures and operators for distributed query processing in our framework. As detailed in Section 7, we make use of a clustering file system, *i.e.*, the Lustre [15] file system. In our system we were not allowed low-level administration, and therefore we could not tune Lustre for optimal performance by imposing *ad-hoc* parameters: we therefore chose a file format that could exploit a parallel distributed access on a shared file system, in particular, we chose the *Hierarchical Data Format* version 5, or HDF5. The HDF5 data format [10] is a binary storage that allows a hierarchical organization of large datasets. It supports platform-independent binary data types, multidimensional arrays, and grouping in order to provide more articulated data structures. In comparison to standard DBMSs, recent developments in the analysis of biological dataset highlighted that databases are effective in dealing with string-based data, whereas management is more difficult for complex numerical structures (cf. Millard et al., Nature [17]). Limits of HDF5 on the top of a Lustre file system is beyond present-day realistic constraints: the maximum archive size is imposed by HDF5, 10^{18} bytes, or 1000 PB.

Permanent Storage. Several data structures have been proposed for (sparse) tensors, *i.e.*, multidimensional arrays [24]. A common representation for sparse matrices is the *Compressed-Row Storage* format, or briefly CRS, with its dual CCS—*Compressed-Column Storage* (cf. [6]). Such matrices with nnz non-zero entries, are represented by means of three different arrays: one of length nnz representing all stored entries, in row-major order for CRS (column-major for CCS); one array of length equal to the number of rows, containing the indexes of the first element of each row (or column); finally, the column index vector of each non-zero element. Literature describes numerous data structures related to CRS, aimed at tensor representation: in essence, elements are stored by *sorting* indexes, and subsequently memorizing index vectors as CRSs, a technique commonly known as *slicing*. It is clear as the *order of sorting* is crucial: being $\mathcal{R}_{ijk}$ a tensor sorted on the i-th coordinate, calculating $\mathcal{R}_{ijk}v_i$ is optimized, but $\mathcal{R}_{ijk}v_k$ is not [2]. Moreover, all CRS descendants suffer from the same drawbacks of their ancestor: they highly depend on the assumption that elements are *evenly distributed among rows*. Moreover, such data structures are *bounded to particular dimensions* of a tensor, or in simpler terms, changing the size of a coordinate—*e.g.*, introducing a new property—is a burdensome operation (cf. [14]).

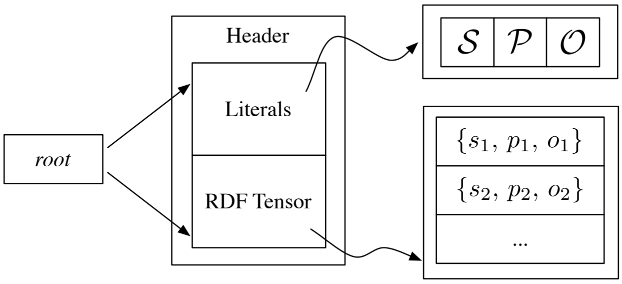


Figure 6: Data storage within the HDF5 file format.

We therefore chose another common data format to model RDF graphs as tensors, the *Coordinate Sparse Tensors* [2], or CST. This format, already introduced in Section 2, memorizes tensors as a list of tuples: we memorize a list of nnz entries, describing the entry value and coordinates, parallel to the description illustrated in Figure 3. The main advantage of this organization is its simplicity and adaptability: it is *order independent* with respect to the RDF tuples, allows fast parallel access to data, requires no particular index sorting on coordinates, and allows run-time dimension changes with the addition of new entries.

As previously said, we chose HDF5 as the hierarchical permanent storage medium. The *root* of the HDF5 storage will contain a header with pointers to two main data structured: the *Literals* list and the *RDF tensor* itself. The former, as exemplified in Figure 6, contains the list of all literals needed by a user to identify objects in the RDF graph: in other words, it incorporates the list of literals and constants found in RDF groups $\mathcal{S}$, $\mathcal{P}$, and $\mathcal{O}$, hence, it implicitly defines $\mathbb{S}$, $\mathbb{P}$, and $\mathbb{O}$. The latter group is the RDF tensor, stored as a list of triples by means of Coordinate Sparse Tensor representation. By definition if a triple is not present in the list, then its associated boolean value is *false*, hence omitted.

Parallel Operations. Our storage exploits the performance boost obtainable by a binary interface and numerical data. We are able, therefore, to *split* data over different processes, so that I/O overhead may be further ameliorated. The CST data structure does not rely on any particular ordering, and therefore given p processes, and being n the number of triples stored in the RDF tensor, we may simply distribute evenly ${}^{n}/_{p}$ 3-tuples on each process, owing to the associative and distributive properties of linear forms. In fact, given $\mathcal{R}_{ijk}v_\ell$, with $\ell \in \{i, j, k\}$, it is perfectly licit to obtain the result as

$$\mathcal{R}_{ijk}v_\ell = \left(\sum_{z=1}^{p} R^z_{ijk}\right) v_\ell = \sum_{z=1}^{p} \left(R^z_{ijk}\, v_\ell\right) , \qquad (1)$$

```
typedef __uint128_t rdft;

// Returns an 128-bit integer from components
inline rdft toStorage(long s, long p, long o)
{
    return (static_cast<rdft>(s) << 0x4E) |
           (static_cast<rdft>(p) << 0x32) |
           (static_cast<rdft>(o));
}

auto it = std::find_if(p.begin(), p.end(),
    [](rdft d){ return d &
                toStorage(42, 0xFFFFFFF, 256); });
```

Figure 7: A representative search with 128-bits integer encoding, searching for a triple matching $\langle \mathbb{S}^{-1}(42), ?x,\ \mathbb{O}^{-1}(256)\,\rangle$.

being $\mathcal{R}^{z}_{ijk}$ the z-th tensor partition, *i.e.*, a set of $^{n}/_{p}$ triples assigned to each process. As the reader may perceive, a parallel implementation of all the operators introduced in the previous sections is straightforward, being inherently data-parallel [3].

We remind that our environment shall handle *highly unstable data-sets*, and as such we do not perform any indexing, as said before. Therefore, each and every computational node in our environment will read a portion of all RDF triples independently of any order, *i.e.*, as they appear in the dataset. Having access to the Lustre distributed file system, each node in the cluster may read its contiguous portion of data, *i.e.*, z-th processor will read $^{n}/_{p}$ triples, with offset equal to $z\,^{n}/_{p}$, with $z \in \mathbb{N} \leq p$ being the processor unique id, and p being equal to the number of available processes. Hence, each process will hold a fragment of the whole tensor $\mathcal{R}$, being the part in itself a valid sparse tensor.

Equation (1) shows that the application of a tensor to a vector may be conducted independently on each process, multiplying the partial tensor $\mathcal{R}^{z}$ with the vector v. In order to reconstruct the complete result $\mathcal{R}\,v$, we shall sum all the contribution computed by each process, an operation that, in distributed terms, is named *reduction*. The reduction operator combines all data from every process with an *associative operation*, in our case, we employ reductions over boolean rings and vector spaces (cf. Algorithm 1), and are carried on communicating among processes using binary trees [22].

Tensor Application. Our implementation has the primary objective of exploiting the underlying hardware for accelerating tensor applications. This paragraph briefly describes the implementation of tensor application described in Section 3.2, and leveraged on each computational node as reported in the previous paragraph.

The tensorial framework we developed relies on the latest ISO C++11 specification, utilizing an unordered vector as the main computational node in-memory data structure. The vector contains all triples stored in a single computational node, encoded as a single *128-bit unsigned integer*; each integer is decomposed bit-by-bit, *i.e.*, interpreted as a sequence of bits representing, in order, $\mathbb{S}$, $\mathbb{P}$, and $\mathbb{O}$. In our implementation, we reserved 50 bits for subject and object, and 28 bits for the property, as detailed in the function `toStorage` in Figure 7. Applying the tensor to a vector falls into one of the four cases exposed in Section 3.2, dependently on each triple pattern's degree of freedom: each DOF case shall multiply the tensor with one or more Dirac deltas (*i.e.*, it is a generalization of Kronecker delta). However, we may conduct those operations simultaneously by scanning the vector for matching triples, encoded in a single 128-bit integer.

Scanning the vector, which is guaranteed to be allocated in a single contiguous block of memory, leverages memory caches and minimizes *cache misses*. In other words, the naïve data structure allows us to employ a simple bit-wise *cache-oblivious* search algorithm [7]. In order to optimize queries, searching is performed by utilizing the bit-wise and operator: a SPARQL triple pattern is encoded as a single 128-bit integer, shifting their numerical values and or-ing them; free variables, as for instance in Figure 7, are represented by a sequence of bit set to 1. Our implementation optimises further comparisons by exploiting CPU-level SSE2/SSE3 instructions, available on every modern processors, and used as accelerators in several computational fields, as for instance bioinformatics or databases. Hence, a triple $\langle s, p, ?x\rangle$ is encoded combining $\mathbb{S}^{-1}(s)$, $\mathbb{P}^{-1}(p)$, and a sequence of 50 bit set in a single 128-bit integer number, treated via XMM 128-bit capable registers.

6. THEORETICAL ANALYSIS

The ensuing paragraphs analyze the theoretical complexity of all the operations involved in SPARQL queries, according to Sections 3 and 4. In the following, we will employ the notation $nnz(M)$, with M being the rank-3 RDF tensor under analysis, denoting the number of its non-zero values—analogously $nnz(v)$ yields the number of non-zero entries of a vector v.

Insertion. The assembly of a sparse tensor requires the basic operation of inserting an element into the list of non-zero values, if not present. The operation has therefore a complexity of $O\left(nnz(M)\right)$, and $O\left(nnz(v)\right)$ for vectors.

Deletion and Update. Such basic actions mimic the above insertion operation, and therefore have an asymptotic complexity of $O\left(nnz(M)\right)$.

Hadamard Product. The Hadamard product of two vectors $u \circ v$ has a complexity of $O\left(nnz(u)\ nnz(v)\right)$.

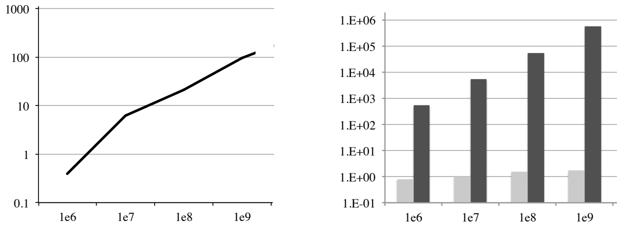
Tensor Application. For a suitable vector v, the tensor application on the ℓ-th dimention $M_{ijk} v_{\ell}$, with $\ell \in \{i, j, k\}$, has asymptotic complexity of $O\left(nnz(M)\right)$, as detailed in [2].

Mapping. Mapping a function on a tensor or a vector, *i.e.*, filtering information, has clearly a linear complexity $O(nnz(M))$ and $O(nnz(v))$. All other non-zero element will be mapped once, and eventually inserted in the result, if the mapping yields a *non-false* value.

Scheduling. The naïve scheduler described in Section 4.1 is optimal. First, let us consider a (computational) *cost function* of a triple pattern. In our environment, where no statistical information about the SPARQL dataset is available, we may assume the degree of freedom of each triplet as an indicator of the supposed computational cost. Let $s^{*} = \{s_1, \ldots, s_t\}$ be the optimal scheduled triple patterns in a SPARQL query, *i.e.*, the scheduling with the minimal cost; if the actual scheduling s differs from s^{*} therefore at least one step i, the algorithm chose a different triple pattern, *i.e.*, $s_i \neq s^{*}_i$, with $dof(s_i) > dof(s^{*}_i)$. However, this contradicts the step 2 of the algorithm presented in Section 4.1: if this were the case, the chosen scheduled triplet would have been s^{*}_i, hence, we have that necessarily $s^{*} = s$.

7. RESULTS

We implemented our framework into TENSORRDF, a C++ system using OpenMPI v1.8 library for answering SPARQL queries over RDF datasets. We performed a series of experiments aimed at



(a) (b)

Figure 8: Data loading times in seconds (a) and query memory footprint expressed in MB (b). Light gray bars refer to system memory overhead, while dark gray ones show data set size.

evaluating the performance of our approach, with the main results detailed in this section.

Benchmark Environment. We deployed TensorRDF on a cluster, wherein each machine is supported by 48 GB DDR3 RAM, 16 CPUs 2.67 GHz Intel Xeon (*i.e.*, each with 4 cores and 8 threads), running Scientific Linux 5.7, with the TORQUE Resource Manager process scheduler. The system is provided with the Lustre file system v2.1, coupled with HDF5 library v1.8.7. The performance of our systems has been measured with respect to data loading, memory footprint, and query execution time with reference to z processes running on z hosts. We evaluated the performance of TensorRDF comparing with centralized triple stores Sesame, Jena-TDB, and BigOWLIM, and open-source systems BitMat [1] and RDF-3X [18], as well as distributed MapReduce-RDF-3X [11], Trinity.RDF [30] and TriAD-SG [8] (i.e., since neither Trinity.RDF and TriAD are openly available, we will refer to the running times reported in [30] and [8]).

In our experiments, we employed the popular LUBM synthetic benchmark (i.e. LUBM-4450 which consists of about 800M triples) and two real-life datasets: DBpedia v3.6, *i.e.*, 200M triples loaded into the official SPARQL endpoint, and BTC-12, the dataset of 2012 Billion Triples Challenge, that is more than 1000M triples. For each dataset we involved a set of test queries. For DBpedia we wrote 25 queries of increasing complexity (available at https://www.dropbox.com/sh/pz0i67s9ohbpb9t/oEGo-J8yui). Such queries involve SELECT SPARQL queries embedding *concatenation* " . ", FILTER, OPTIONAL and UNION operators. We use such dataset for comparison with centralized approaches. In this case, we set up TensorRDF on a single machine. Referring to BTC-12, we exploit the test queries defined in [18]; in both LUBM-4450 and BTC-12 we have SELECT SPARQL queries involving only *concatenation*. This two last datasets are used for comparison with distributed approaches.

Loading and Memory Footprint. Referring to data management, we are able to achieve three main goals. First, we are able to perform loading *without any particular relational schema*, when compared to triple store approaches, where a schema coupled with appropriate indexes have to be maintained by the system. Second, owing to the flexibility of CST we are capable of *modifying substantially the tensor dimension*, *i.e.*, introducing novel literals in either RDF sets is a trivial operation: whereas a DBMS must perform a re-indexing, we may carry this operation without any additional overhead. As last objective, owing to the distributivity and associativity properties of our theoretical model, we are able to distribute data and computational power over different hosts, allowing also parallel access to the data. In this case, we refer to a 12-server cluster deployment. Data loading times are 45, 110 and 130 seconds for DBpedia, LUBM-4450, and BTC-12, respectively. In particular, as showed in Figure 8(a), data loading times are 0.395, 6.194, 21.068, and 129.699 seconds, for all examined dimensions in BTC-12. Another significant advantage of our system relies in memory consumption. In particular, always referring to a 12-server cluster deployment, the overall memory overhead needed to maintain a distributed tensor representation of RDF data almost constant, and amounts to circa 1 MB of RAM. Referring to BTC-12, the total distributed memory consumption for our tests were 549.3 MB, 5.391, 44.121, and 332.918 GB for all examined dimensions; both memory overhead and RAM occupation are depicted in Figure 8(b). On average, all triple store systems require a data space 10 times greater, BitMat 5 times greater, RDF-3X, Trinity.RDF and TriAD-SG 2-3 times greater.

Query Analysis. We ran the queries ten times and measured the average response time (including the I/O times) in ms. On disk-based systems, we performed both *cold-cache* and *warm-cache* experiments. Figure 9 illustrates the response times on DBpedia in a 1-server cluster (*i.e.*, centralized environment). On average, Sesame and Jena-TDB perform poorly, BigOWLIM and BitMat better, and RDF-3X is competitive. TensorRDF outperforms all competitors, in particular RDF-3X by a large margin. TensorRDF is 18 times better than RDF-3X, 128 times on the maximum (*i.e.*, Q21). In particular the queries involving OPTIONAL and UNION operators (*e.g.*, Q20) require the most complex computation: triple stores, *i.e.*, BigOWLIM, Sesame and Jena-TDB, depend on the physical organization of indexes, not always matching the joins between patterns. RDF-3X provides a permutation of all combinations of indexes on subject, property and object of a triple to improve efficiency. However queries, embedding OPTIONAL and UNION operators in a graph pattern with a considerable size, require complex joins between huge number of triples (*i.e.*, Q20) that compromises the performance. On the other hand, we exploit the sparsity of our tensor and compute in parallel map functions, tensor applications, and Hadamard products. Another strong point of our system is a very low consumption of memory for query execution, due to the *sparse matrix* representation of tensors and vectors. Figure 10 shows the memory usage (KB) to query DBpedia in a 1-server cluster. On the average, all queries (also the most complex) require very few bytes of memory (*i.e.*, dozens of KBytes), whereas all competitors require dozens of MB.

As distributed systems, we measured times for TensorRDF, TriAD-SG (i.e., TriAD using Summary Graph), Trinity.RDF and MapReduce-RDF-3X (simply MR-RDF-3X) on a 12-server cluster with 1GBit LAN connection for both LUBM-4450 and BTC-12. We highlight that we used a set of SELECT queries embedding only *concatenation*, on which competitors exploit own physical indexing in a profitable way. Figure 11 presents the results, showing that our system performs 9 times better than MR-RDF-3X and 5 times better thanTrinity.RDF for LUBM-4450, while 100 times better than MR-RDF-3X and 1,5 times better thanTrinity.RDF for BTC-12. TriAD-SG is the most competitive system: however for queries non selective (i.e. LUBM-4450) our system is comparable to TriAD-SG, while for selective queries (i.e. BTC-12) our system outperforms TriAD-SG. This is due by exploiting the algebraic properties of our tensorial representation that allows us to process in parallel triples in small chunks and by embedding DOF evaluation to speed-up selective triples execution. Referring to memory consumption, querying LUBM-4450 and BTC-12 presents a behavior (*i.e.*, dozens of KBytes) similar to DBpedia, while competitors dozens of MB. For space constraints, we do not report the diagram.

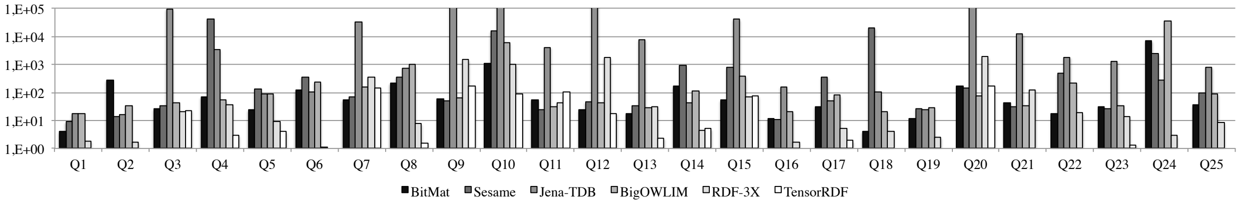


Figure 9: Times on DBpedia in ms: different gray scale bars refer each system, *i.e.*, from BitMat (black), to TENSORRDF (white). Response time of TENSORRDF for Q1, Q2, Q6, Q16, Q18, Q19, Q21, Q22 and Q25 was less than 1 ms.

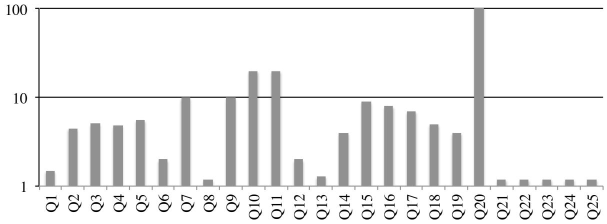
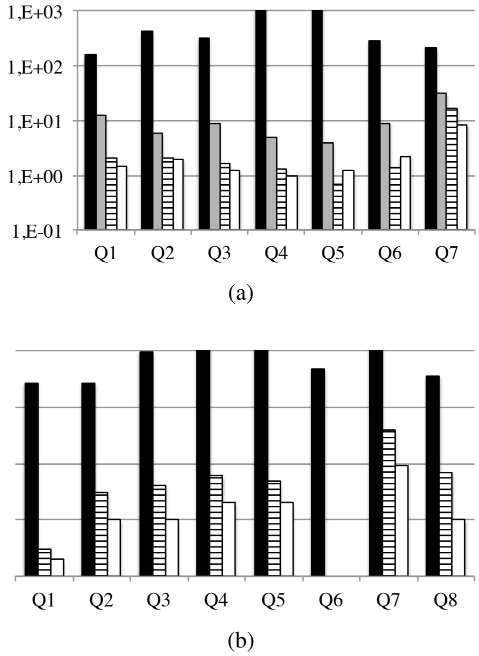


Figure 10: Memory Usage to query DBPEDIA in KB.

We also performed *warm-cache* experiments, but we are unable to report them due to space constraints; however while TENSORRDF improves performance from milliseconds to microseconds (*e.g.*, from 1 ms to 0.1 μs), the other competitors improves performance in milliseconds magnitude (*e.g.*, from 100 ms to 1 ms). As last experiment, we tested TENSORRDF's scalability, reported in Figure 12 with a limited representative number of queries (*i.e.*, Q3, Q6 and Q7 in BTC-12 since they are the most complex). As the dimension of our problem increases, from 500MB to 300GB, the time increases from approximately 10^{-3} ms, to 10^{1} ms for the largest dimension, *i.e.*, for a number of triples of 10^{9}.



(a)

(b)

Figure 11: Response Times in ms on LUBM-4450 (a), and BTC-12 (b), for MR-RDF-3X (in black), Trinity.RDF (in gray), TriAD-SG (striped filled) and TENSORRDF (in white).

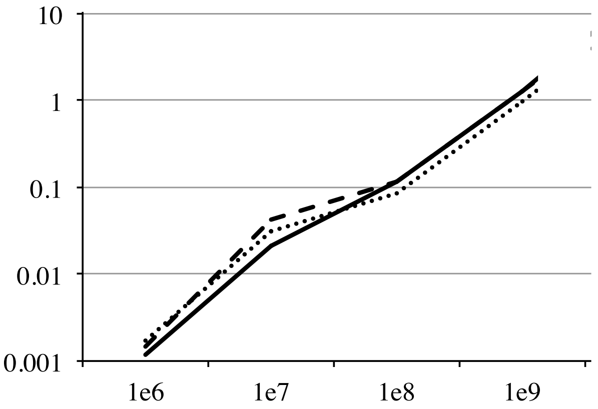


Figure 12: Scalability on BTC-12: times (ms) are plotted against number of triples. Solid, dotted and dashed lines refer respectively to Q4, Q7, and Q8.

8. RELATED WORKS

Existing systems for the management of Semantic-Web data can be discussed according to two major issues: *storage* and *querying*. Considering the storage, two main approaches can be identified: developing native storage systems with ad-hoc optimizations, and making use of traditional DBMSs (such as relational and object-oriented). Native storage systems (such as OWLIM, or RDF-3X [18]) are more efficient in terms of load and update time, whereas the adoption of mature data management systems exploit consolidate and effective optimizations. Indeed, native approaches need re-thinking query optimization and transaction processing techniques. However, the number of required self-joins makes this approach impractical, and the optimizations introduced to overcome this problem have proven to be query-dependent, or to introduce significant computational overhead (cf. [4]).

On the querying side, current research in SPARQL pattern processing (cf. [18, 8] and [27]) focuses on optimizing the class of so-called *conjunctive* patterns (possibly with filters) under the assumption that these patterns are more commonly used than the others. Nevertheless, a keen observation of SPARQL queries from real-life logs [12]showed that *non-conjunctive* queries are employed in non-negligible numbers, providing detailed statistics. An interesting approach was given in [1] by Atre et al., which start from a dense (*i.e.*, not sparse) tensorial representation, and generate all possible combinations of two dimensional matrices of relations, named BitMats, discarding some pairings such as Object-Property "since based on [our] experience, usage of those BitMats is rare"; finally, they *compress* row-wise with a RLE scheme, amounting on a total of $2|\mathcal{P}| + |\mathcal{S}| + |\mathcal{O}|$ matrices. Query computation require significant workload and post-processing.

On the other hand, our approach exploits algebraic properties for both storage and computation, and owing to the DOF sorting, is able to schedule triplets in an efficient order; additionally, our framework may deal with non-conjunctive queries and data change. As illustrated in Section 7, all the above approaches are optimized for centralized analysis requiring significant amount of resources, both in terms of memory and storage. In opposite, Trinity.RDF [30] and TriAD [8] exploit in-memory frameworks to provide efficient general-purpose query processing on RDF in a distributed environment. However, as shown in Section 7, the efficiency on such proposals is strictly depending on the logical and physical organization of data, and on the complexity of the query. Differently from the other approaches, TENSORRDF provides a *general-purpose* storage policy for RDF graphs. Our approach exploits linear algebra and tensor calculus principles to define an abstract model, independent from any logical or physical organization, allowing RDF dataset to change comfortably and to distribute computation over several hosts, without any *a priori* knowledge about RDF dataset or querying statistics.

9. CONCLUSIONS AND FUTURE WORK

We have presented an abstract algebraic framework for the efficient and effective analysis of RDF data. Our approach leverages tensorial calculus, proposing a general model that exhibits a great flexibility with queries, at diverse granularity and complexity levels (*i.e.*, both *conjunctive* and *non-conjunctive* patterns with filters). Experimental results proved our method efficient when compared to recent approaches, yielding the requested outcomes in memory constrained architectures. For future developments we are investigating the introduction of reasoning capabilities, along with a thorough deployment in highly distributed Cloud environments.

10. REFERENCES

[1] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for rdf data. In *WWW*, pages 41–50, 2010.

[2] B. W. Bader and T. G. Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2007.

[3] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1997.

[4] S. M. D. J. Abadi, A. Marcus and K. Hollenbach. SW-Store: a vertically partitioned DBMS for semantic web data management. *VLDB J.*, 18(2):385–406, 2009.

[5] I. N. Davidson, S. Gilpin, O. T. Carmichael, and P. B. Walker. Network discovery via constrained tensor analysis of fmri data. In *KDD*, pages 194–202, 2013.

[6] T. A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, 2006.

[7] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–297, 1999.

[8] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In *SIGMOD*, pages 289–300, 2014.

[9] M. Hammoud, D. A. Rabbou, R. Nouri, S. Beheshti, and S. Sakr. DREAM: distributed RDF engine with adaptive query planner and minimal communication. *PVLDB*, 8(6):654–665, 2015.

[10] G. Heber. HDF5 meets, challenges, and complements the DBMS. In *XLDB*, 2011.

[11] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.

[12] M. H. K Möller, R. Cyganiak, S. Handschuh, and G. Grimnes. Learning from linked open data usage: Patterns & metrics. In *Web Science Conference*, 2010.

[13] T. G. Kolda and J. Sun. Scalable tensor decompositions for multi-aspect data mining. In *ICDM*, pages 363–372, 2008.

[14] C.-Y. Lin, Y.-C. Chung, and J.-S. Liu. Efficient data compression methods for multidimensional sparse array operations based on the ekmr scheme. *IEEE Trans. Comput.*, 52:1640–1646, 2003.

[15] M. W. Margo, P. A. Kovatch, P. Andrews, and B. Banister. An analysis of state-of-the-art parallel file systems for linux. In *5th International Conference on Linux Clusters: The HPC Revolution 2004*, 2004.

[16] K. Maruhashi, F. Guo, and C. Faloutsos. Multiaspectforensics: Pattern mining on large-scale heterogeneous networks with tensor analysis. In *ASONAM*, pages 203–210, 2011.

[17] B. L. Millard, M. Niepel, M. P. Menden, J. L. Muhlich, and P. K. Sorger. Adaptive informatics for multifactorial and high-content biological data. *Nature Methods*, 8(6):487–492, 2011.

[18] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD*, pages 627–640, 2009.

[19] N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, and N. Koziris. H_2rdf+: an efficient data management system for big RDF graphs. In *SIGMOD*, pages 909–912, 2014.

[20] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *Trans. Database Syst.*, 34(3), 2009.

[21] F. Picalausa and S. Vansummeren. What are real sparql queries like? In *SWIM - SIGMOD Workshop*, page 7, 2011.

[22] R. Rabenseifner. Optimization of collective reduction operations. In *Computational Science-ICCS 2004*, pages 1–9. Springer, 2004.

[23] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *ICDT*, pages 4–33, 2010.

[24] M. P. Sears, B. W. Bader, and T. G. Kolda. Parallel implementation of tensor decompositions for large data analysis. In *SIAM*, 2009.

[25] J. Sun, S. Papadimitriou, and P. S. Yu. Window-based tensor analysis on high-dimensional and multi-aspect streams. In *ICDM*, pages 1076–1080, 2006.

[26] J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: dynamic tensor analysis. In *KDD*, pages 374–383, 2006.

[27] M.-E. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra, and A. Polleres. Efficiently joining group patterns in sparql queries. In *ESWC*, pages 228–242, 2010.

[28] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[29] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. Triplebit: a fast and compact system for large scale rdf data. *PVLDB*, 6(7):517–528, 2013.

[30] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *PVLDB*, pages 265–276, 2013.

[31] S. Zhang, S. Li, and J. Yang. Gaddi: distance index based subgraph matching in biological networks. In *EDBT*, pages 192–203, 2009.

Motivation-Aware Task Assignment in Crowdsourcing

Julien Pilourdault Sihem Amer-Yahia
Université Grenoble Alpes, CNRS, LIG, France
firstname.lastname@imag.fr

Dongwon Lee
Penn State University, USA
dongwon@psu.edu

Senjuti Basu Roy
NJIT, USA
senjutib@njit.edu

ABSTRACT

We investigate how to leverage the notion of *motivation* in assigning tasks to workers and improving the performance of a crowdsourcing system. In particular, we propose to model motivation as the balance between *task diversity*–i.e., the difference in skills among the tasks to complete, and *task payment*–i.e., the difference between how much a chosen task offers to pay and how much other available tasks pay. We propose to test different task assignment strategies: (1) RELEVANCE, a strategy that assigns *matching* tasks, i.e., those that fit a worker's profile, (2) DIVERSITY, a strategy that chooses *matching* and *diverse* tasks, and (3) DIV-PAY, a strategy that selects *matching* tasks that offer *the best compromise between diversity and payment.* For each strategy, we study multiple iterations *where tasks are re-assigned to workers as their motivation evolves.* At each iteration, RELEVANCE and DIVERSITY assign tasks to a worker from an available pool of filtered tasks. DIV-PAY, on the other hand, estimates each worker's motivation on-the-fly at each iteration, and uses it to assign tasks to the worker. Our empirical experiments study the impact of each strategy on overall performance. We examine both requester-centric and worker-centric performance dimensions and find that different strategies prevail for different dimensions. In particular, RELEVANCE offers the best task throughput while DIV-PAY achieves the best outcome quality.

1. INTRODUCTION

Crowdsourcing has become a popular framework to solve problems that are often hard for computers but easy for humans. Examples of crowdsourcing tasks include sentiment analysis in text, extracting information from images, and transcribing audio records. Despite recent successes, however, one of longstanding challenges in crowdsourcing is *task completion*–i.e., some tasks remain only partially completed or some workers do not work at full capacity. Studies have indicated that *workers' motivation* [25] plays a key role in task completion, especially since micro-tasks usually have very small monetary compensation. Therefore, it becomes increasingly important to *understand and model workers' motivation appropriately in the task assignment step.* In this paper, therefore, we study motivation in crowdsourcing and how to leverage it to assign tasks to workers. Our goal is to reach a better understanding of how to model motivation by studying its impact on different performance dimensions in crowdsourcing.

Existing literature has extensively studied how to perform task assignment to workers on crowdsourcing platforms [8, 17, 18, 23, 26]. Task assignment considers goals such as maximizing the quality of completed tasks, or minimizing task cost and latency to complete tasks. More recently, some research has reported noticeable improvement in task outcome quality when human factors, such as workers' skills and expected wage, were used in assigning tasks to workers [23, 26].

Yet, even when tasks are perfectly matched and assigned to workers initially, an important longstanding problem is *how to keep motivating workers who are not well-engaged in completing assigned tasks.* A recent ethnomethodological study on Turker Nation [22] argued that in order to maintain the attractiveness of crowdsourcing platforms, it is critical to enable *worker-centric optimization.* To address this problem, some existing work focused on incentivizing workers for long-lasting tasks [5, 19] or entertaining workers during task completion [7]. Moreover, recent studies have experimentally demonstrated the importance of intrinsic motivation in task completion [25]. While effective to some extent, these methods do not perceive task completion as an iterative process within which workers' motivation evolves, neither do they model that in the task assignment process. In this work, we advocate the need to *account for the evolution of workers' motivation* as workers complete tasks and capture that evolution in task assignment.

Our idea. Organization studies have explored worker motivation in physical workplaces since 70's [14]. Recently, some new efforts have examined and experimentally explored motivation on crowdsourcing platforms such as Amazon Mechanical Turk (AMT) [20, 25]. They have largely come to the conclusion that the motivation model developed in physical workplaces was also applicable in virtual marketplaces such as AMT.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.23

Modeling workers' motivation is not obvious. While some workers may be driven by fun and enjoyment, others may look to advance their human capital, or increase their compensation. In fact, there are more than 13 factors that could be used to model motivation according to [20] (e.g., *task payment*, *task diversity*, *task autonomy*, *task identity*, *human capital advancement*, *pastime*). In addition, in a given session, a worker's motivation for a task may also depend on tasks that she has already completed and on other available tasks. In this work, as a first attempt to model workers' motivation and account for it in task assignment, we decided to focus on two factors: (1) *task diversity*, that is akin to skill variety, and (2) *task payment.* We believe that these two factors are good representatives of the spectrum of factors. Other factors will be examined in future.

Our formalization is grounded in the theory of work redesign [14]. The choice of diversity and payment allows us to clearly distinguish between "intrinsic" factors (e.g., how interested a worker is in the task's content) and "extrinsic" factors (e.g., how much the task pays). We can therefore use our formalization to verify which of intrinsic or extrinsic factors influence a worker's performance during task completion. Our formalization serves as a basis to define the **motivation-aware task assignment** (`Mata`) as a constrained optimization problem. Specifically, given a set of available tasks and a set of workers who are not working at full capacity, we identify which tasks are to be re-assigned to which worker, considering the worker's motivation. The worker-centric *and* adaptive nature of our problem make it novel. Indeed, while learning workers' skills in a crowdsourcing platform has been addressed before (e.g., [23]), leveraging those factors on-the-fly in a task assignment has not been addressed. There also exists a range of studies on online (iterative) task assignment but they consider only quality [8, 17, 18, 29] or budget and deadlines [11] in their objective. In fact, they rely on measuring the effectiveness of workers (e.g., in terms of *accuracy* [8, 29]) to adapt their assignment policy and do not consider the motivation of workers.

In contrast, our work considers motivation as a *first-class* factor in the objective function. Hence, none of the techniques proposed in the previous studies are applicable to our problem. To the best of our knowledge, our work is the first to propose to periodically revisit task assignment to workers by modeling and monitoring their motivation.

Our contributions. We propose to first formalize motivation factors that directly affect task completion. We then define our motivation-aware task assignment problem (`Mata`). We show that `Mata` is NP-hard using a reduction from the maximum dispersion problem (`MaxSumDisp`) [6, 10, 16, 24]. To solve our problem and isolate the effect of different dimensions on workers' motivation, we design and compare three task assignment strategies: (1) RELEVANCE, a strategy that chooses tasks that *match* a worker's profile, (2) DIVERSITY, a strategy that chooses *matching and diverse* tasks, and (3) DIV-PAY, a strategy that selects *matching* tasks with *the best compromise between diversity and payment.* DIV-PAY requires to observe workers as they complete tasks, estimate their motivation dynamically, and suggest the next most appropriate tasks. DIV-PAY is a $\frac{1}{2}$-approximation algorithm that uses a solution from the maximum diversification problem (`MaxSumDiv`), a general case of `MaxSumDisp`.

For each strategy, we study multiple iterations where tasks are re-assigned to workers. In order to compare our strategies, we develop a framework to hire workers from AMT and monitor them during task completion. In order to examine the effect of task diversity, we select $158,018$ micro-tasks released by CrowdFlower. Those tasks belong to 22 different kinds ranging from tweet classification to extracting information from news and assessing the sentiment of a piece of text. We measure common requester-centric dimensions such as task throughput (i.e., the number of tasks completed in multiple iterations per unit of time) and outcome quality with respect to a ground truth. We also measure dimensions that are considered both requester-centric and worker-centric, namely, worker retention (i.e., the number of workers who completed tasks) and payment. Worker motivation is measured as a worker-centric dimension.

Our empirical validation shows that different strategies prevail for different dimensions. RELEVANCE outperforms both DIV-PAY and DIVERSITY on task throughput and worker retention. However, DIV-PAY outperforms the other strategies on outcome quality. Workers completed more tasks and stayed longer when they were assigned tasks with RELEVANCE. That could be explained by the fact that very little context switching is required from workers in the case of RELEVANCE (since tasks are both relevant to the worker's profile and are potentially very similar to each other). DIVERSITY, on the other hand, is slightly inferior to DIV-PAY. That leads to the conclusion that diversity alone is not satisfactory as workers also pay attention to payment. The fact that DIV-PAY achieves the best outcome quality proves the need to actively monitor workers' motivation and incorporate it in task assignment. *Indeed, even if they are faster at completing similar tasks and stay longer in the system when tasks are relevant and not diverse, workers provide a higher-quality outcome for tasks that optimize their motivation, i.e., those chosen to achieve a balance between diversity and payment.* This confirms the need for worker-centric and adaptive approaches in crowdsourcing.

Paper organization. Section 2 formalizes the problem of motivation-aware crowdsourcing. Section 3 describes our three task assignment strategies. Section 4 reports performance results. Section 5 contains the related work. Finally, Section 6 summarizes our findings and their implications, and discusses possible future directions.

2. DATA MODEL AND PROBLEM

In this section, we first describe our model for tasks and motivation factors. Then, we formalize the motivation-aware task assignment problem. Table 1 summarizes important notations used throughout the paper.

2.1 Tasks and Workers

We consider a set of tasks $\mathcal{T} = \{t_1, \ldots, t_n\}$, a set of workers $\mathcal{W} = \{w_1, \ldots, w_p\}$ and a set of skill keywords $\mathcal{S} = \{s_1, \ldots, s_m\}$.

Notation	Definition
$\mathcal{T}$	a set of tasks $\{t_1, \dots, t_n\}$
$\mathcal{W}$	a set of workers $\{w_1, \dots, w_p\}$
$\mathcal{S}$	a set of skill keywords $\{s_1, \dots, s_m\}$
$d(t_k, t_l)$	pairwise task diversity between two tasks
$\mathcal{T}_w^i$	tasks assigned to worker w at iteration i
$TD(\mathcal{T}')$	task diversity of a set of tasks $\mathcal{T}' \subseteq \mathcal{T}$
$TP(\mathcal{T}')$	task payment of a set of tasks $\mathcal{T}' \subseteq \mathcal{T}$
α_w	a worker w's relative importance between *task diversity* and *task payment*
$motiv_w(\mathcal{T}_w^i)$	the expected motivation of worker w on tasks $\mathcal{T}_w^i$
X_{max}	maximum number of tasks assigned to a worker
$matches(w, t)$	returns TRUE if the keywords of worker w *match* the keywords of task t

Table 1: A summary of important notations.

Tasks. A task t is represented by a vector $\langle t(s_1), t(s_2), \dots, t(s_m), c_t\rangle$. For all $j \in [\![1, m]\!], t(s_j)$ is a Boolean value that denotes the presence or absence of skill keyword s_j in task t. The reward c_t is given to a worker who completes t. In our model, skill keywords may be interpreted as interests or qualifications, thereby allowing to capture a variety of tasks.

Workers. A worker w is represented by a vector $w = \langle w(s_1), \dots, w(s_m)\rangle$. For all $j \in [\![1, m]\!], w(s_j)$ is a Boolean value capturing the interest of w in the skill keyword s_j.

EXAMPLE 1. Table 2 shows an example with 3 tasks, 2 workers and 5 skills. For instance, t_1 is characterized by a vector $\langle \texttt{true}, \texttt{true}, \texttt{false}, \texttt{false}, \texttt{false}, 0.01\rangle$: it is an audio transcription task with a \$0.01 reward, and it is described by skill keywords "*audio*" and "*English*". w_1 is a worker who expresses interest in tasks that feature the keywords "*audio*" and "*tagging*". We could suppose that only workers covering all task skills are qualified to complete a task. In this example, w_1 would only qualify for task t_2, while w_2 would qualify for both t_1 and t_3. □

2.2 Motivation Factors

Related work from the social sciences [20] on worker motivation in crowdsourcing includes 13 factors. The 6 most important ones are *Payment, Task Autonomy, Skill Variety, Task Identity, Human Capital Advancement, Pastime.* In this work, as a first attempt to model workers' motivation and account for it in task assignment, we focus on two factors: *task payment* and *task diversity*, that is akin to skill variety. Each factor is computed using a function that returns a motivation score. The choice of payment and diversity allows us to clearly distinguish between extrinsic motivation (payment) and intrinsic motivation (diversity) and offer enough variety in their values to study subtle differences in motivation. In addition, compared to other dimensions, only these two dimensions are most relevant in micro-tasks and in typical labor markets, such as Amazon Mechanical Turk. How to incorporate the remaining factors in modeling motivation is left to future work.

	audio	*English*	*French*	*review*	*tagging*	*reward* (\$)
t_1	✓	✓				0.01
t_2					✓	0.03
t_3			✓	✓		0.09
w_1	✓				✓	N/A
w_2	✓	✓	✓	✓		N/A

Table 2: Example of tasks and workers

Task Diversity. We denote the *pairwise task diversity* between two tasks t_k and t_l by $d(t_k, t_l)$. Pairwise task diversity essentially measures the aggregated differences of skills between two tasks. We ignore task reward in this definition. In our setting, we use the Jaccard similarity function $J()$ to define $d()$ as follows:

$$d(t_k, t_l) = 1 - J(\langle t_k(s_1), \dots, t_k(s_m)\rangle, \langle t_l(s_1), \dots, t_l(s_m)\rangle)$$

$d()$ is a metric and verifies the triangular inequality. We aim to be general and we do not fix one particular definition of $d()$ here. Instead, we allow any distance function (e.g., Euclidean distance, Jaro distance) as long as it verifies the triangular inequality. The *Task diversity* $TD(\mathcal{T}')$ of a set of tasks $\mathcal{T}' \subseteq \mathcal{T}$ is captured by aggregating the pairwise distances in $\mathcal{T}'$:

$$TD(\mathcal{T}') = \sum_{(t_k, t_l) \in \mathcal{T}'} d(t_k, t_l) \tag{1}$$

Task Payment. The total task payment of a set of tasks $\mathcal{T}' \subseteq \mathcal{T}$ is the sum of individual task payments in $\mathcal{T}'$:

$$TP(\mathcal{T}') = \frac{1}{\max_{t \in \mathcal{T}} c_t} \times \sum_{t \in \mathcal{T}'} c_t \tag{2}$$

The denominator $\max_{t \in \mathcal{T}} c_t$ normalizes each member of the sum in the interval $[0, 1]$.

2.3 Modeling a Worker's Motivation

We advocate a multi-step approach where the set of tasks assigned to a worker are revisited at each step in order to best fit the worker's motivation. At each iteration i, a worker w is assigned a new set of tasks $\mathcal{T}_w^i$. We wish to determine the best set $\mathcal{T}_w^i$ at each iteration i.

To capture the expected motivation of worker w on tasks in $\mathcal{T}_w^i$, we define a function $motiv_w^i$ as a linear[1] combination of diversity and payment of tasks in $\mathcal{T}_w^i$:

$$\begin{aligned} &motiv_w^i(\mathcal{T}_w^i) = \\ &2\alpha_w^i \times TD(\mathcal{T}_w^i) + (|\mathcal{T}_w^i| - 1)(1 - \alpha_w^i) \times TP(\mathcal{T}_w^i) \end{aligned} \tag{3}$$

[1] Note that we define the function as a linear formula between diversity and payment of a task, instead of a more complex non-linear formula, since a linear formula is likely to give rise to algorithms with theoretical guarantees as we show later, and is easier to interpret/explain.

α_w^i is a worker-specific parameter that reflects the relative importance between *task diversity* and *task payment*. We normalize the two components of the function with the factors $(|\mathcal{T}_w^i| - 1)$ and 2, since the first part of the sum counts $\frac{|\mathcal{T}_w^i|(|\mathcal{T}_w^i|-1)}{2}$ numbers and the second part $|\mathcal{T}_w^i|$ numbers [13]. We aim to accurately compute α_w^i that represents the compromise a worker w is looking for in choosing tasks to complete at each iteration i.

EXAMPLE 2. A worker w_1 with $\alpha_w^i = 0.1$ would be interested more in high-paying tasks with similar keywords (i.e., less diversity). This worker w_1 would choose tasks with a variety of keywords only if the payment is high enough. On the other hand, a worker w_2 with $\alpha_w^i = 0.9$ would be more motivated by task diversity. □

2.4 Problem

Now, we formally define the motivation-aware task assignment problem, `Mata`, as follows:

Problem 1 (Motivation-Aware Task Assignment) At each iteration i, and for each worker $w \in \mathcal{W}$, choose a subset of tasks $\mathcal{T}_w^i \subseteq \mathcal{T}$ such that:

$$\begin{aligned} \textbf{max} \quad & motiv_w^i(\mathcal{T}_w^i) & \\ \textit{such that} \quad & \forall t \in \mathcal{T}_w^i \ matches(w,t) & (C_1) \\ & |\mathcal{T}_w^i\ | \leq X_{max} & (C_2) \end{aligned} \qquad \blacksquare$$

The function $matches(w,t)$ in constraint C_1 returns `true` if the task t *matches* worker w. We can use various definitions for $matches(w,t)$. For instance, we can define $matches(w,t) =$ `true` iff the skill keywords of w and t are identical. In our setting, we suppose that $matches(w,t)$ captures how well the skill keywords of w *cover* the skill keywords of t. This allows us to capture cases where w *matches* t only if w expresses interest in at least 50% of the skill keywords of t. X_{max} is used in constraint C_2 to avoid assigning too many tasks to workers with varied interests, and reflects the ability of a worker to explore only a few tasks at a time (akin to limiting Web search results). Throughout this paper, we will suppose that each time we solve the `Mata` problem for a given worker w, w matches at least X_{max} tasks. Thus, given that the objective function is positive and monotonically increasing, w will be assigned *exactly* X_{max} tasks. That is a realistic assumption when X_{max} is chosen to be reasonably small (e.g., 20) in a context where we have a large collection of tasks to assign.

It is also important to note that the `Mata` problem considers each worker *independently*. When a worker w requires a new set of tasks $\mathcal{T}_w^i$, `Mata` is solved and tasks in $\mathcal{T}_w^i$ are dropped from $\mathcal{T}$. Thus, a task is assigned to at most one worker.

3. OUR APPROACHES

In order to study the effect of different dimensions in our problem, now, we explore approaches that exploit different objectives in the `Mata` problem. First, we design RELEVANCE, a diversity and payment-agnostic strategy. This strategy focuses on assigning to workers tasks that best match their interests. Second, we present DIV-PAY, that optimizes the objective function of the `Mata` problem. DIV-PAY is hence both diversity and payment-aware. Third, we present DIVERSITY, that focuses only on assigning diverse tasks to workers and is hence payment-agnostic.

Algorithm 1 RELEVANCE

```
Input: $\mathcal{T}$, $w, X_{max}, i$
Output: $\mathcal{T}_w^i$
  $\mathcal{T}_w^i \leftarrow \emptyset$
  $\mathcal{T}_{match(w)} \leftarrow \{t \in \mathcal{T} \mid matches(w,t)\}$
  while $|\mathcal{T}_w^i| < X_{max}$
      $\mathcal{T}_w^i \leftarrow \mathcal{T}_w^i \cup \{\text{nextRandomTask}(\mathcal{T}_{match(w)} \setminus \mathcal{T}_w^i)\}$
  return $\mathcal{T}_w^i$
```

3.1 Relevance strategy

We first propose the RELEVANCE approach (Algorithm 1), that assigns X_{max} random tasks that match workers' interests. RELEVANCE enforces constraints C_1 and C_2 and ignores task diversity and task payment. At each iteration i and for each worker w, RELEVANCE (i) filters the tasks $\mathcal{T}_{match(w)}$ that match w and (ii) selects randomly X_{max} tasks among $\mathcal{T}_{match(w)}$. In this strategy, a worker's motivation is therefore interpreted as matching her interests.

3.2 Diversity and payment-aware strategy

We present DIV-PAY, a strategy that is both diversity and payment-aware. DIV-PAY relies on both the on-the-fly estimation of a worker's motivation, and the online iterative task assignment. The motivation of a worker w is captured in the value of α_w^i, which represents the compromise a worker w is looking for in choosing tasks to complete at each iteration i. We first describe our approach to computing α_w^i. We then describe the task assignment algorithm that aims to solve the complete `Mata` problem.

3.2.1 Computing α_w^i

At each iteration i, we aim to learn α_w^i by leveraging tasks $t \in \mathcal{T}_w^{i-1}$ completed by worker w. Here, $\mathcal{T}_w^{i-1}$ refers to the tasks that were assigned to w in the previous iteration $(i-1)$ and that were presented to w as the set of available tasks. Let us consider the j-th task selected by worker w in $\mathcal{T}_w^{i-1}$. Each time when a worker selects a task t_j, we want to learn her preference for task diversity and task payment. To that purpose, we define α_w^{ij} to capture the compromise between skill variety and task payment made by the worker w when choosing t_j during iteration i. Our goal is to leverage a collection of such "micro-observations". First, we aim to capture each α_w^{ij}. Then, we aim to aggregate all α_w^{ij} to compute α_w^i.

We first show how to capture each α_w^{ij}. We start by considering each motivation factor independently. We suppose that when worker w chooses task t_j, she has already chosen tasks $\{t_1, \ldots, t_{j-1}\}$ where $j-1 \in [\![1, |\mathcal{T}_w^{i-1}|]\!]$.

Task Diversity. Equation 4 shows how we capture the gain in task diversity that a worker w seeks when picking a task

Algorithm 2 DIV-PAY

```
Input: $\mathcal{T}$, $w, X_{max}, i, \mathcal{T}_w^{i-1}, \{t_1, \ldots, t_J\}$ tasks completed in iteration $i-1$
Output: $\mathcal{T}_w^i$
$\alpha_w^i \leftarrow \text{avg}_{k\in[\![1,J]\!]}\, \alpha_w^{ij}$
$\mathcal{T}_{match(w)} \leftarrow \{t \in \mathcal{T} \mid matches(w,t)\}$
$\mathcal{T}_w^i \leftarrow$ GREEDY$(\mathcal{T}_{match(w)}, X_{max}, w)$
return $\mathcal{T}_w^i$
```

t_j in the remaining tasks $\mathcal{T}_w^{i-1} \setminus \{t_1, \ldots, t_{j-1}\}$.

$$\Delta\,TD(t_j) = \frac{\sum\limits_{k\in 1,\ldots,j-1} d(t_j, t_k)}{\max\limits_{t_{k'}\in\mathcal{T}_w^{i-1}\setminus\{t_1,\ldots,t_{j-1}\}} \sum\limits_{k\in 1,\ldots,j-1} d(t_{k'}, t_k)} \tag{4}$$

The numerator is the marginal gain in diversity when w selects a task t_j. The denominator reflects the maximum possible marginal gain when w selects a task in the remaining tasks $\mathcal{T}_w^{i-1} \setminus \{t_1, \ldots, t_{j-1}\}$.

Task Payment. We compute the list of all *different* task payments in $\mathcal{T}_w^{i-1} \setminus \{t_1, \ldots, t_{j-1}\}$ and sort it by descending order. Suppose that this list counts R elements and that $r(t_j)$ is the rank of c_{t_j} in this list (if c_{t_j} is the highest then $r(t_j) = 1$). We define *TP-Rank*$(t_j) \in [0,1]$ such that *TP-Rank*$(t_j) = 1$ *iff* t_j has the highest payment (0 if it has the lowest payment):

$$\textit{TP-Rank}(t_j) = 1 - \frac{r(c_{t_j}) - 1}{R - 1} \tag{5}$$

Equation 5 captures the willingness of w to choose tasks that pay highly among the available tasks.

EXAMPLE 3. Suppose that $\mathcal{T}_w^{i-1} \setminus \{t_1, \ldots, t_{j-1}\} = \{t_5, t_6, t_7, t_8\}$ with $c_{t_5} = \$0.03, c_{t_6} = c_{t_7} = \$0.02, c_{t_8} = \$0.04$. A worker w selects t_5, which has the second highest task payment among the remaining tasks. We obtain *TP-Rank*$(t_5) = 1 - \frac{2-1}{3-1} = 0.5$. □

We have defined how to capture the importance of each factor. We now need to define α_w^{ij}, that captures the compromise between task diversity and task payment that worker w seeks when selecting task t_j. We set:

$$\alpha_w^{ij} = \frac{\Delta\,TD(t_j) + 1 - \textit{TP-Rank}(t_j)}{2} \tag{6}$$

α_w^{ij} is defined as the average of $\Delta\,TD(t_j)$ and $1-\textit{TP-Rank}(t_j)$. The asymmetry comes from the fact that the higher α_w^i is, the lower is the importance of the task payment factor. We can observe that if both $\Delta\,TD(t_j)$ and $1 - \textit{TP-Rank}(t_j)$ return the same value, α_w^{ij} will be equal to 0.5.

Having defined each α_w^{ij}, we are now ready to capture α_w^i. Suppose that during iteration $i-1$ worker w chose J tasks where $J \leq |\mathcal{T}_w^{i-1}|$. We compute α_w^i as the average of all α_w^{ij}:

$$\alpha_w^i = \underset{k\in[\![1,J]\!]}{\text{avg}}\, \alpha_w^{ij} \tag{7}$$

Algorithm 3 GREEDY [4]

```
Input: $\mathcal{T}_{match(w)}, X_{max}, w, i$
Output: $\mathcal{T}_w^i$
$\mathcal{T}_w^i \leftarrow \emptyset$
while $|\mathcal{T}_w^i| < X_{max}$
    $t \leftarrow \arg\max_{t'\in\mathcal{T}_{match(w)}\setminus\mathcal{T}_w^i} g(\mathcal{T}_w^i, t')$
    $\mathcal{T}_w^i \leftarrow \mathcal{T}_w^i \cup \{t\}$
return $\mathcal{T}_w^i$
```

3.2.2 Assigning Tasks

At each iteration i, the DIV-PAY strategy aims to solve the Mata problem for each worker. We first show that the Mata problem is NP-hard. Then, we present the DIV-PAY algorithm that returns a solution with an approximation ratio of 2 for the Mata problem.

Complexity. Intuitively, the Mata problem is difficult since its objective function includes the sum of pairwise distances, a common feature in several well-known NP-hard problems. In particular, Mata is closely related to the maximum dispersion problem (MaxSumDisp) [6, 10, 16, 24].

Theorem 1 *The motivation-aware task assignment problem (Mata) is NP-hard.* ■

PROOF. At each iteration and for each worker, the decision version of Mata is as follows. *Instance*: tasks $\mathcal{T}$, worker w and her α_w^i, X_{max} and an objective value Z. *Question*: is there a set $\mathcal{T}_w^i \subseteq \mathcal{T}$ such that C_1 is satisfied, $|\mathcal{T}_w^i| = X_{max}$ and $motiv_w^i(\mathcal{T}_w^i) \geq Z$? Mata $\in$ *NP* since a non-deterministic algorithm needs only to guess a set $\mathcal{T}_w$ and it can verify the question in polynomial time.

Reduction of Max Dispersion. To prove the NP-hardness, we consider the maximum sum dispersion problem (MaxSumDisp) [6, 10, 16, 24] (also known as *Maximum Edge Subgraph*)[2]. The decision version of this problem is as follows. *Instance*: a complete weighted graph $G = (V, E, \omega)$, an integer $k \in [2, |V|]$, an objective value Y. *Question*: Is there $V' \subseteq V$ such that $|V'| = k$ and $\sum_{v_1,v_2\in V'} \omega(v_1, v_2) \geq Y$?

Note that MaxSumDisp is well-known to be NP-hard [4, 6, 10, 24] using a reduction from MaxClique [12]. Because a non-deterministic algorithm can guess a solution V' and easily verify it in polynomial time, MaxSumDisp $\in$ *NP*. Thus, MaxSumDisp is also NP-Complete. The reduction works as follows: each vertex $v \in V$ is mapped to a task $t_v \in \mathcal{T}$. The weight of an edge $(v_1, v_2) \in E$ is mapped to skill variety between two tasks: $\omega(\{v_1, v_2\}) = 2 * d(t_{v_1}, t_{v_2})$. We consider that $\alpha_w^i = 1$. We set $X_{max} = k$ and $Z = Y$. This creates an instance of Mata in polynomial time. This instance has the objective function $2 * \sum_{t_k,t_l\in\mathcal{T}_w^i} d(t_k, t_l)$. MaxSumDisp has a solution if and only if this instance of Mata has a solution. This proves the NP-hardness. □

Algorithm. Because it is an NP-hard problem, Mata is prohibitively expensive to solve on large instances. In our scenario, however, response time is important since Mata has to

[2] http://www.nada.kth.se/˜viggo/wwwcompendium/node46.html

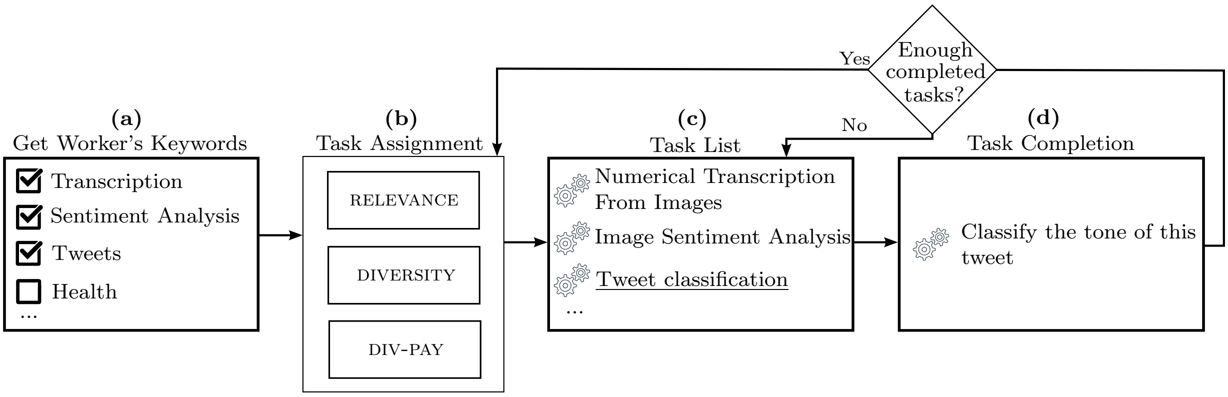


Figure 1: Workflow of our motivation-aware platform

be solved *online*, at each iteration i. The good news is that approximation algorithms exist for some related problems, such as MaxSumDisp [15, 16, 24] if the distance d satisfies the triangle inequality. The assumption that the distance obeys triangle inequality is not an overstretch, as many real world distances satisfy this assumption [3]. We consider that the pairwise task diversity is a metric and follows triangle inequality.

We adapt an existing algorithm for the maximum diversification problem MaxSumDiv (which includes MaxSumDisp as a special case). We design DIV-PAY (Algorithm 2) that assigns a set of tasks $\mathcal{T}_w^i$ to a worker w. First, DIV-PAY captures the motivation of the worker in the value of α_w^i. Then DIV-PAY computes a set of matching tasks (line 2) and runs GREEDY that returns $\mathcal{T}_w^i$ (line 3). GREEDY (Algorithm 3) is a $\frac{1}{2}$-approximation algorithm for the MaxSumDiv problem [4]. In the MaxSumDiv problem, the objective is to find a set S of p elements that maximizes

$$\lambda \sum_{(u,v)\in S} d(u,v) + f(S)$$

λ is a weight parameter, $f(S)$ is a normalized, monotone submodular function measuring the value of S and $d()$ is a distance function evaluating the diversity between two elements. Since Mata simplifies to finding a set of size *exactly* X_{max}, the objective function can be rewritten as:

$$\begin{aligned} &motiv_w^i(\mathcal{T}_w^i) = \\ &2\alpha_w^i \times TD(\mathcal{T}_w^i) + (X_{max}-1)(1-\alpha_w^i) \times TP(\mathcal{T}_w^i) \end{aligned}$$

Now, we map our problem to the MaxSumDiv problem by setting $f(\mathcal{T}_w^i) = (X_{max}-1) \times (1-\alpha_w^i) \times TP(\mathcal{T}_w^i)$, $\lambda = 2\alpha_w^i$ and $p = X_{max}$. It can be easily seen that f is normalized ($f(\emptyset) = 0$). f is monotone since $\forall \mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{T}$ s.t. $\mathcal{T}_1 \subseteq \mathcal{T}_2$ we have $f(\mathcal{T}_1) \leq f(\mathcal{T}_2)$. It is also submodular since $\forall \mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{T}$ s.t. $\mathcal{T}_1 \subseteq \mathcal{T}_2$ and $\forall t \in \mathcal{T}$, we have:

$$\begin{aligned} f(\mathcal{T}_1 \cup \{t\}) - f(\mathcal{T}_1) &= (X_{max}-1)(1-\alpha_w^i) \times \frac{1}{\max_{t'\in\mathcal{T}} c_{t'}} \times c_t \\ &= f(\mathcal{T}_2 \cup \{t\}) - f(\mathcal{T}_2) \end{aligned}$$

At each iteration, GREEDY inserts in $\mathcal{T}_w^i$ the task t that maximizes the function $g(\mathcal{T}_w^i, t) = \frac{1}{2}(f(\mathcal{T}_w^i \cup \{t\}) - f(\mathcal{T}_w^i)) + \lambda \sum_{t'\in\mathcal{T}_w^i} d(t,t')$ which is equal to $g(\mathcal{T}_w^i, t) = (X_{max}-1)(1-\alpha_w^i)TP(\{t\})/2 + 2\alpha_w^i \sum_{t'\in\mathcal{T}_w^i} d(t,t')$.

Algorithm 4 DIVERSITY

```
Input: $\mathcal{T}$, $w, X_{max}, i$
Output: $\mathcal{T}_w^i$
1: $\alpha_w^i \leftarrow 1$
2: $\mathcal{T}_{match(w)} \leftarrow \{t \in \mathcal{T} \mid matches(w,t)\}$
3: $\mathcal{T}_w^i \leftarrow$ GREEDY($\mathcal{T}_{match(w)}, X_{max}, w$)
4: return $\mathcal{T}_w^i$
```

We run GREEDY using tasks that verify the constraint C_1 (Algorithm 2, line 2), thus the algorithm returns a correct solution for the Mata problem. Because GREEDY is a $\frac{1}{2}$-approximation algorithm for the MaxSumDiv problem, DIV-PAY is a $\frac{1}{2}$-approximation for the Mata problem. Borodin et al. [4] observe that the GREEDY algorithm runs in time linear in the number of input elements when the desired size of the set is a constant. In our setting, we can conclude that DIV-PAY runs in $\mathcal{O}(X_{max} * |\mathcal{T}|)$ time.

One may wish to extend the motivation model used in Mata. We observe that the performance guarantee and the running time of GREEDY holds as long as our objective function has the form $\lambda \sum_{(u,v)\in S} d(u,v) + f(S)$ where f is a normalized, monotone and submodular function.

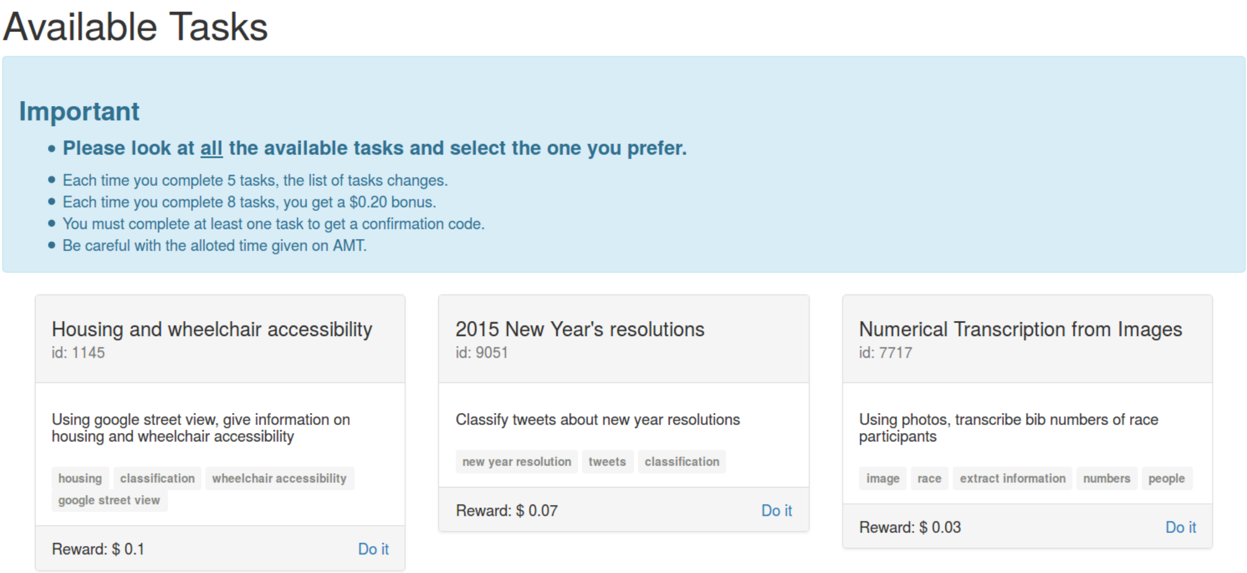
3.3 Diversity strategy

We propose DIVERSITY (Algorithm 4), a strategy that is diversity-aware and payment-agnostic. DIVERSITY considers a variant of the Mata problem where the objective includes only the task diversity sum. DIVERSITY employs GREEDY as a subroutine with $\alpha_w^i = 1$ at every iteration. We can follow the same reasoning exposed for Mata: constraint C_1 is enforced on line 2 and GREEDY is a $\frac{1}{2}$-approximation for the considered problem, so DIVERSITY is a $\frac{1}{2}$-approximation for this variant of the Mata problem.

4. EMPIRICAL VALIDATION

4.1 Workflow

We developed a web application to support motivation-aware crowdsourcing. Figure 1 illustrates a work session within our application. First, we get the interests of the worker w (Figure 1a). Then, we assign w a set of tasks (Figure 1b). Here, we can employ one of the three strate-



Available Tasks

Important

- **Please look at all the available tasks and select the one you prefer.**
- Each time you complete 5 tasks, the list of tasks changes.
- Each time you complete 8 tasks, you get a $0.20 bonus.
- You must complete at least one task to get a confirmation code.
- Be careful with the alloted time given on AMT.

Housing and wheelchair accessibility
id: 1145

Using google street view, give information on housing and wheelchair accessibility

housing classification wheelchair accessibility google street view

Reward: $ 0.1 Do it

2015 New Year's resolutions
id: 9051

Classify tweets about new year resolutions

new year resolution tweets classification

Reward: $ 0.07 Do it

Numerical Transcription from Images
id: 7717

Using photos, transcribe bib numbers of race participants

image race extract information numbers people

Reward: $ 0.03 Do it

Figure 2: Example screenshot of user interface – e.g., task grid

gies RELEVANCE, DIV-PAY or DIVERSITY. Then, the worker is presented a list of tasks (Figure 1c). She chooses a task and completes it (Figure 1d). If she has completed less than a pre-determined number of the X_{max} tasks, she is presented the same set of tasks again, except the tasks that were already completed. If she has completed enough tasks, another task assignment iteration is executed.The rationale behind imposing a minimum number of completed tasks before reiteration is to get a sufficient amount of input to accurately estimate α_w^i for each worker.

For the strategies RELEVANCE and DIVERSITY, we run the according algorithm at each iteration. For the strategy DIV-PAY, we need to consider the first iteration ($i = 1$) where w arrives for the first time on our platform. In this case, we cannot compute her α_w^1 since she has not yet completed tasks. In this first iteration, task assignment uses RELEVANCE as a cold-start assignment strategy. We aim to learn w's preference between diversity and payment using a strategy that does not favor any factor. Our rationale is to get an accurate estimation of α_w^1. On the next iterations, since w has already completed tasks, we run DIV-PAY. We compute her α_w^i and return the new set of tasks $\mathcal{T}_w^i$.

4.2 Settings

4.2.1 *Tasks*

We used a set of 158,018 micro-tasks released by Crowdflower [1]. It includes 22 different kinds of tasks, featuring for instance tweet classification, searching information on the web, transcription of images, sentiment analysis, entity resolution or extracting information from news. Each different kind of task is assigned a set of keywords that best describe its content and a reward, ranging from $0.01 to $0.12. Considered tasks are *micro-tasks* (they took on average 23s to be completed). We set payment proportional to the expected completion time.

4.2.2 *Task assignment*

We conducted experiments with all task assignment strategies, RELEVANCE, DIV-PAY, and DIVERSITY. We adapted the RELEVANCE strategy because the distribution of tasks is not uniform in our dataset (there are kinds of tasks that are over represented). The random task selection was achieved by first selecting a random *kind* of task, and then selecting a random task of this particular kind. We set $X_{max} = 20$ and imposed that 5 tasks must be completed before running another iteration. We set $matches(w, t) = \texttt{true}$ *iff* w is interested by at least 10% of the keywords of task t. Workers were asked to provide at least 6 keywords. We also verified the response time of our algorithms: any approach returned a solution in a few milliseconds upon a worker request. This makes our approaches suitable for an online setting: new workers and tasks can be easily handled by recomputing assignments from scratch.

4.2.3 *Workers and Payment*

We published 30 HITs on Amazon Mechanical Turk (AMT) to recruit workers. Each HIT corresponds to a work session on our platform. When a worker accept a HIT, she is asked to visit our web application. On our platform, she completes multiple tasks. When she terminates her work session, she get a verification code. Then, she paste the code on AMT and submit the HIT for payment. Each HIT may be submitted by at most 1 worker. Our HITs were completed by 23 different workers. We assigned 10 HITs for each task assignment strategy.

The HIT reward was set to $0.1. Each worker was granted a bonus equivalent to the total reward of the tasks she completed. To encourage workers who completed many tasks, we granted them a $0.2 bonus each time they completed 8 tasks. We required workers to have previously completed at least 200 HITs that were approved, and to have an approval rate above 80%. We also required HITs to be completed within 20 minutes: our rationale is to encourage workers to choose quickly tasks that they prefer.

4.2.4 User Interface

We conducted a first set of experiments where the $\mathcal{T}_w^i$ were displayed as a ranked list. We observed that most workers selected the top task first, completed it, and walked down the list in order. This created a bias and defeated our purpose: observing workers making choices based on their motivation. In order to reduce the effect of ranking, we changed the interface by showing a grid with 3 tasks per row (Figure 2). That mitigated the effect of ranking and workers stopped choosing the task in their order of appearance. We used the grid-based presentation in all our experiments.

4.2.5 Evaluation measures

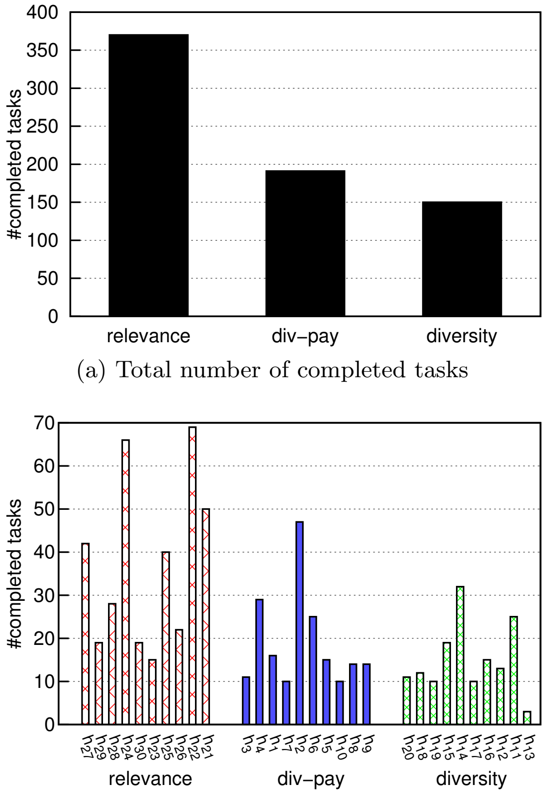
We evaluate our task assignment strategies using various measures. First, we consider *requester-centric* measures. Those include the *total number of completed tasks* across all iterations, and the *number of completed tasks per minute*, and *task throughput*, i.e., the number of completed tasks per work session. The higher the throughput, the faster a requester can have crowdwork completed. However, this measure does not reveal the quality of crowdwork. Hence, we also measure the *quality* of workers' contributions. Then, we consider measures that can be seen as both *requester* and *worker-centric*. That is the case for *task payment* and *worker retention* that quantifies the number of workers who completed tasks and captures the willingness of workers to work on our tasks. Finally, we measure *worker motivation*, a *worker-centric* dimension that quantifies workers' preferences between task diversity and task payment.

4.3 Results

23 different workers completed 711 tasks in 30 work sessions. On average, each worker spent 13 minutes to submit the HIT and completed 23.7 tasks. On average, 73% of workers chose fewer than 10 keywords (6 is the minimum possible).

4.3.1 Number of Completed tasks

Figure 3a presents the total number of completed tasks. Overall, RELEVANCE clearly outperforms DIV-PAY, which is slightly better than DIVERSITY. Figure 3b details the number of completed tasks for each work session h_k, $k \in [\![1, 30]\!]$. We observe that with RELEVANCE, 5 sessions had more than 40 completed tasks. With DIV-PAY and DIVERSITY, most workers completed fewer than 30 tasks. Figure 4 presents task throughput (i.e., number of completed tasks per min). We considered the total time spent on our application, including the time spent selecting a task to complete. The total time was higher with RELEVANCE (157 min) than with DIV-PAY (127 min). However, workers who were assigned tasks with RELEVANCE were more efficient (2.35 tasks/min vs 1.5 tasks/min). This could be explained by the fact that very little context switching is required from workers in the case of RELEVANCE (since tasks are both relevant to the worker and are potentially very similar to each other). DIVERSITY on the other hand, is slightly inferior to DIV-PAY. That leads us to the conclusion that workers did not necessarily appreciate diverse tasks, possibly for context-switching reasons. DIV-PAY slightly outperformed DIVERSITY, showing that including task payment as a motivating factor improves task throughput. While task throughput is a good indicator of the speed at which tasks are completed, it does not reveal the quality of crowdwork.



(a) Total number of completed tasks

(b) Number of completed tasks per work session

Figure 3: Number of completed tasks

4.3.2 Quality

For each kind of task, we sampled 50% of completed tasks and we manually evaluated their ground truth. We chose tasks for which defining a ground truth was not controversial (e.g., a task that asks for the presence or not of a pattern on an image). Then, we compared each worker's contribution to a task to its ground truth. Figure 5 presents the percentage of tasks that were correctly completed for each strategy. We observe that workers performed better with DIV-PAY (73% of correct answers) than with other strategies (DIVERSITY: 64%, RELEVANCE: 67%). This shows that assigning tasks that best match workers' compromise between task payment and task diversity encourages them to produce better answers. We observe that considering only task diversity (DIVERSITY) is not efficient. Including task payment is therefore important.

4.3.3 Worker retention

We now evaluate worker retention, a dimension that qualifies as both requester-centric and worker-centric. Figure 6a shows worker retention as the percentage of work sessions (vertical axis) that ended after x tasks were completed

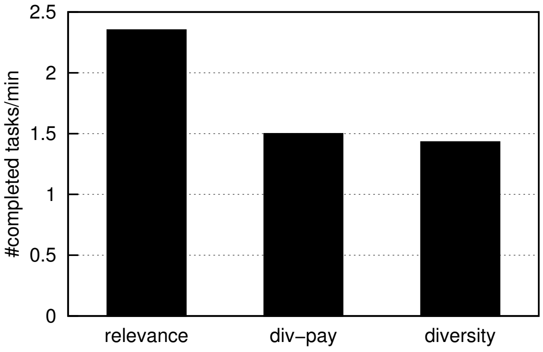


Figure 4: Task throughput

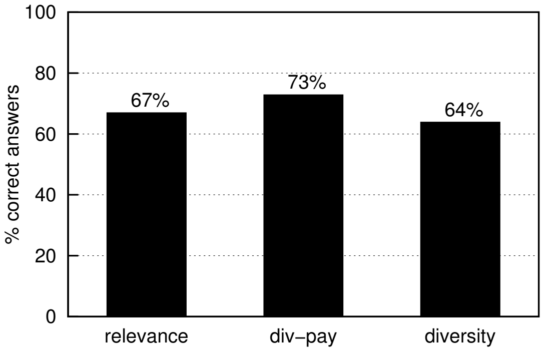
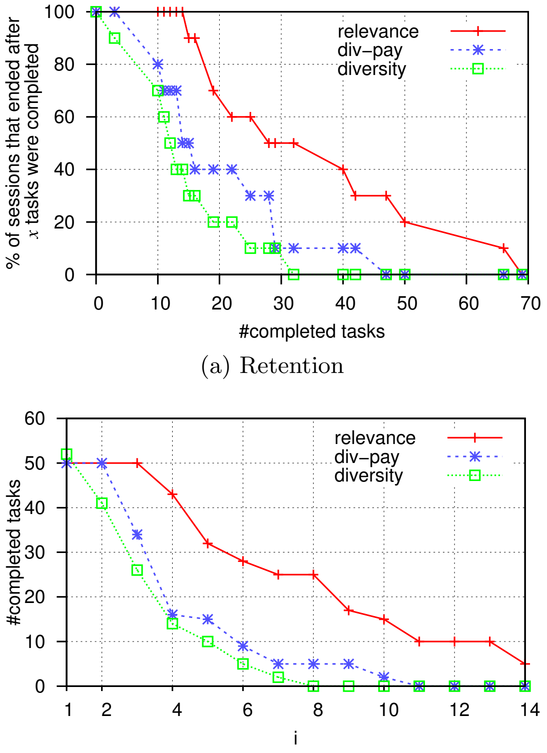


Figure 5: Evaluation of crowdwork quality



(a) Retention

(b) Number of completed tasks per iteration

Figure 6: Worker retention and number of completed tasks per iteration.

(horizontal axis). We find that workers stayed longer and completed more tasks when they were assigned tasks using RELEVANCE, hence this approach clearly outperforms DIV-PAY and DIVERSITY. Figure 6b supports this observation: more iterations were performed by workers with RELEVANCE. Although the number of completed tasks is roughly the same with all approaches on the first 2 iterations, this number fell quickly for DIV-PAY and DIVERSITY when $i > 2$. We also observe that DIV-PAY has a better worker retention than DIVERSITY. A plausible explanation is that workers are most comfortable completing similar tasks in a row. Therefore, they stay longer. They are least comfortable completing tasks with very different skills and tend to leave earlier. However, given that the quality of crowdwork reaches its best with DIV-PAY, we can say that optimizing for task relevance alone does not provide the best outcome quality even if workers are retained longer in the system.

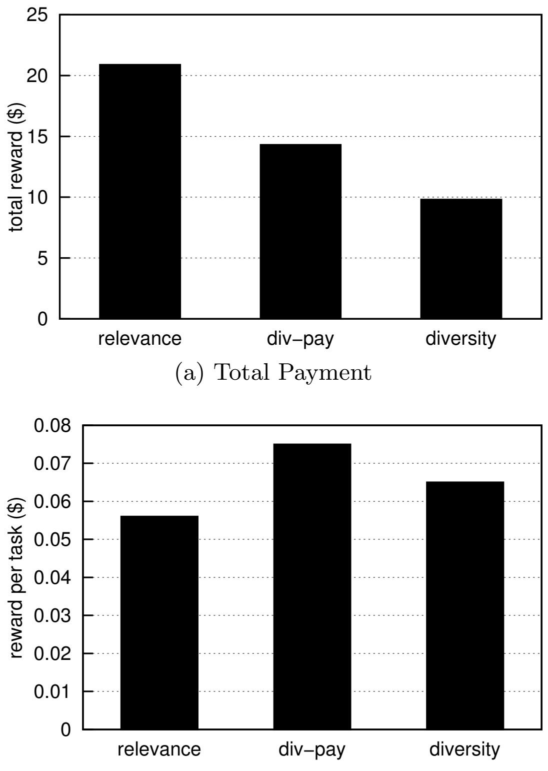
4.3.4 Task Payment

Task payment is the other measure that qualifies as both *requester-centric* and *worker-centric*. Indeed, both requesters and workers look for a fair treatment when it comes to compensation. Requesters look to obtain high-quality contributions at a reasonable rate, and workers expect to be adequately paid for their efforts. Figure 7 presents the total task payment and the average payment per completed task for each strategy. The total payment (Figure 7a) is greater with RELEVANCE than with other approaches. This could be expected given the number of completed tasks (Section 4.3.1). However, the average task payment (Figure 7b) is the greatest with DIV-PAY. That is explained by the fact that DIV-PAY is the only strategy that is payment-aware. Thus, it is likely to assign higher-paying tasks to workers that prefer task payment over task diversity.

4.3.5 Workers' motivation

We now turn to workers and study their motivation in detail. In order to make a fair comparison, we compute α_w^i for each strategy and for each iteration $i \geq 2$ (even if it is only used by DIV-PAY). Figure 8 shows the values of α_w^i for each work session h_k, $k \in 1 \ldots 30$. We omit session h_{13} (with DIVERSITY approach) where only 3 tasks were completed. We observe that in most work sessions, α_w^i oscillates around 0.5. Given the definition of α_w^i, this value indicates that most workers do not *steadily* favor task diversity over task payment. This is particularly observable on long work sessions in Figure 8a, where tasks were assigned using RELEVANCE. Figure 9 shows the distribution of α_w^i. It supports our observation: most workers do not make sharp choices. Most of the computed α_w^i values (72%) are in the interval $[0.3, 0.7]$, meaning that most workers do not favor task diversity over task payment, nor do they favor payment over diversity.

However, we do observe some sharp preferences for some workers. For instance, the worker in session h_2 (Figure 8b) clearly favored high-paying tasks. She completed 1.6 *different* tasks at each iteration (maximum possible: 5) that have an average reward of \$0.11 (maximum possible reward: \$0.12). Hence, her α_w^i was close to 0. Since she was assigned



(a) Total Payment

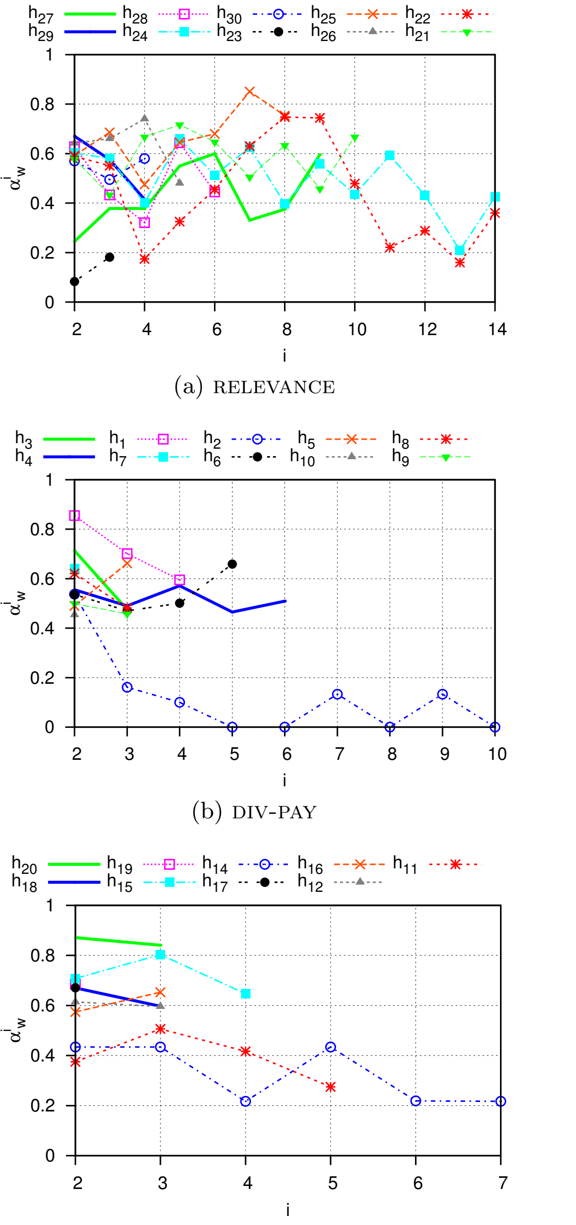
(b) Payment per task

Figure 7: Task payment

tasks using DIV-PAY, she received high-paying tasks. On the other hand, the worker in session h_{25} (Figure 8a) favored task diversity (her α_w^i is close to 0.8). She completed 4.12 *different* tasks at each iteration, that paid $0.05 on average. This shows that our formulation allowed to accurately capture workers' preferences between task diversity and task payment.

4.4 Summary of Results and Discussion

We now summarize our results and provide a rationale for why different strategies prevail for different measures. Let us first consider *requester-centric* measures. We observe that RELEVANCE is the strategy that provides the best *task throughput*. This could be explained by the fact that RELEVANCE requires less effort from workers than DIVERSITY and DIV-PAY. Indeed, since RELEVANCE is based on selecting tasks that best match a worker's profile and since a worker's profile is quite homogeneous, tasks recommended by RELEVANCE are quite similar to each other. Therefore, a worker does not do much context switching between tasks and is hence faster overall. We also observe that DIV-PAY slightly outperforms DIVERSITY on task throughput. That shows the importance of task payment as an incentive. Results are different if we consider *crowdwork quality*. DIV-PAY is the strategy that obtains the best quality, followed by RELEVANCE. DIV-PAY is the only strategy that is both adaptive and motivation-aware: this contributes to providing a better incentive to workers. Quality comes at a price though: DIV-PAY is the strategy where the average *task payment* among completed tasks is the highest and it does not provide the



(a) RELEVANCE

(b) DIV-PAY

(c) DIVERSITY

Figure 8: Evolution of α_w^i (h_k is k-th work session)

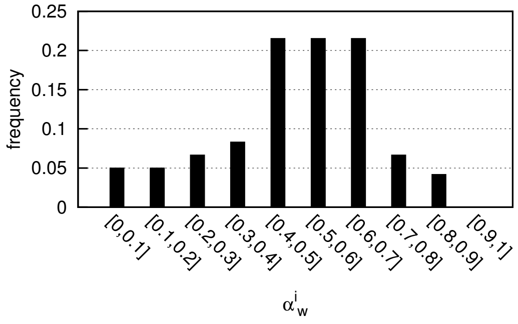


Figure 9: Distribution of α_w^i

highest task throughput (it is better than DIVERSITY but worse than RELEVANCE). Thus, depending on the platform, one should study trade-offs between these strategies when designing task assignment algorithms.

If we consider *worker-centric* measures, we observe that DIV-PAY is the best strategy for *task payment* since it rewarded workers higher. This could be expected since it is the only payment-aware strategy. However, *worker retention* is better with RELEVANCE. Workers performed longer work sessions with RELEVANCE than with other strategies. This finding is related to the fact that most workers do not have a clear preference for task diversity or task payment. They prefer tasks that match their interests and require fewer context switching, hence they did not necessarily stay longer when task diversity or task payment were favored (with DIV-PAY or DIVERSITY). We also observe *workers' motivation* and we notice that some workers carefully choose task diversity or task payment. In that case, we could accurately capture their preferences with appropriate α_w^i values. That allowed DIV-PAY to slightly outperform DIVERSITY on both task throughput and worker retention.

5. RELATED WORK

Task Assignment in Crowdsourcing. Task assignment in crowdsourcing was largely studied. Previous studies notably include the design of *adaptive* algorithms, that focus on maximizing the quality of crowdwork [8, 17, 18, 29]. For instance, Fan et al. [8] leverage the similarity between tasks and the past answers of workers to design an adaptive algorithm that aims at maximizing the accuracy of crowdwork. Ho et al. [17] study an online setting, where the workers who are going to arrive on the platform have unknown skill. They design an algorithm where the skill level of sampled workers is observed and is leveraged to assign all other workers to tasks. None of these studies includes motivation factors in their model. Other investigations focused on dynamically adjusting the task reward [9, 11] so as to satisfy a deadline or a budget constraint. They modeled the willingness of a worker to choose a task as a task acceptance probability featuring task reward as a parameter. These studies do not focus on task assignment as workers self-appoint themselves to tasks and they do not include task diversity in their model. Recently, Rahman et al. [23] focused on assigning tasks to groups of *diverse workers* in collaborative crowdsourcing. Moreover, Wu et al. aim at finding sets of workers with diverse *opinions* [28]. None of those studies assigns *diverse tasks* to workers or includes motivation factors. Moreover, Rahman et al. [23] consider a setting which is not *adaptive*: task assignment does not leverage previous answers to improve the main objective.

Motivating Workers. A range of studies point out the importance of suitably motivating workers in crowdsourcing [2, 21, 22]. Obviously, reward is an important factor, and crowdsourcing platform should follow some guidelines that would solve wage issues [2]. Kittur et al. [21] underlines the interest of designing frameworks that include incentive schemes other than financial ones. In particular, they notice that a system should "*achieve both effective task completion and worker satisfaction*". Worker motivation was first studied in physical workplaces [14]. Recent studies [20] investigated the importance of 13 motivation factors for workers on Amazon Mechanical Turk. Although task payment remains the most important factor, Kaufmann et al. [20] point out that workers are also interested in *skill variety* or *task autonomy*.

Some efforts were driven towards experimenting motivation factors in crowdsourcing [5, 7, 25, 27]. In a recent study [7], Dai et al. inserted *diversions* in the workflow such that workers were presented with some entertainment contents between two task completions. Dai et al. showed that such a motivational scheme improved worker retention. Chandler and Kapelner [5] conducted experiments showing that workers perceiving the "meaningfulness" of task improved throughput without degrading quality. Another study [25] assessed the effect of extrinsic and intrinsic motivation factors. They demonstrated that workers were more accurate on meaningful tasks posted by a non-profit organization than on tasks posted by a private firm and less explicit about their outcome. This suggested that intrinsic factors help improve quality of crowdwork. Shaw et al. [27] assessed 14 incentives schemes and found that incentives based on worker-to-worker comparisons yield better crowdwork quality. None of the above studies leverages motivation factors to optimize task assignment, and thus they do not tackle the motivation-aware task assignment problem.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented three different formulations of task assignment algorithms and compared three motivation-aware task assignment strategies: RELEVANCE, DIVERSITY, and DIV-PAY. Both RELEVANCE and DIVERSITY are based on matching tasks to a worker's interests, while DIV-PAY relies on assigning tasks to workers based on their observed motivation. This work builds on the premise that workers may look for a balance between intrinsic motivation, modeled as task diversity, and extrinsic motivation, modeled as task payment.

In practice, our experiments show that different strategies prevail. RELEVANCE offers the best task throughput and worker retention since it requires less context switching for workers (i.e., all tasks are similar). DIV-PAY, however, has the best outcome quality, since it allows workers to achieve the best compromise between fun and compensation. This last observation is important as it confirms that even when workers are slower at executing tasks and when they spend less time on a platform, optimizing for their motivation impacts task outcome quality positively. There are also a few cases where DIV-PAY outperforms DIVERSITY on task throughput and worker retention, implying the need to account for payment in addition to diversity. Those are the cases of workers whose preference between diversity and payment are sharply expressed as they choose tasks to complete. Our results highlight the importance to understand the evolving motivation of workers on a crowdsourcing platform and the importance to measure both requester-centric and worker-centric dimensions.

In the future, we would like to investigate the possibility of making the platform transparent by showing to workers what the system learned about them and letting them provide explicit feedback. We will also investigate the impact of other motivation factors such as "social signaling" and "advancing human capital" with respect to measures such as task throughput and worker retention.

References

[1] Crowdflower - data for everyone. https://www.crowdflower.com/data-for-everyone/.

[2] B. B. Bederson and A. J. Quinn. Web workers unite! addressing challenges of online laborers. In *CHI*, pages 97–106, 2011.

[3] A. Besicovitch. On a general metric property of summable functions. *Journal of the London Mathematical Society*, 1(2):120–128, 1926.

[4] A. Borodin, H. C. Lee, and Y. Ye. Max-sum diversification, monotone submodular functions and dynamic updates. In *PODS*, pages 155–166, 2012.

[5] D. Chandler and A. Kapelner. Breaking monotony with meaning: Motivation in crowdsourcing markets. *CoRR*, abs/1210.0962, 2012.

[6] B. Chandra and M. M. Halldórsson. Approximation algorithms for dispersion problems. *J. Algorithms*, 38(2):438–465, 2001.

[7] P. Dai, J. M. Rzeszotarski, P. Paritosh, and E. H. Chi. And now for something completely different: Improving crowdsourcing workflows with micro-diversions. In *ACM CSCW*, pages 628–638, 2015.

[8] J. Fan, G. Li, B. C. Ooi, K.-l. Tan, and J. Feng. icrowd: An adaptive crowdsourcing framework. In *SIGMOD*, pages 1015–1030, 2015.

[9] S. Faradani, B. Hartmann, and P. G. Ipeirotis. What's the right price? pricing tasks for finishing on time. In *AAAI*, 2011.

[10] S. P. Fekete and H. Meijer. Maximum dispersion and geometric maximum weight cliques. *CoRR*, cs.DS/0310037, 2003.

[11] Y. Gao and A. G. Parameswaran. Finish them!: Pricing algorithms for human computation. *PVLDB*, 7(14):1965–1976, 2014.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[13] S. Gollapudi and A. Sharma. An axiomatic approach for result diversification. In *WWW*, pages 381–390, 2009.

[14] J. Hackman and G. R. Oldham. Motivation through the design of work: Test of a theory. *Organizational Behavior and Human Performance*, 16(22):250–279, 1976.

[15] R. Hassin and S. Rubinstein. An improved approximation algorithm for the metric maximum clustering problem with given cluster sizes. *Inf. Process. Lett.*, 98(3):92–95, 2006.

[16] R. Hassin, S. Rubinstein, and A. Tamir. Approximation algorithms for maximum dispersion. *Oper. Res. Lett.*, 21(3):133–137, 1997.

[17] C. Ho, S. Jabbari, and J. W. Vaughan. Adaptive task assignment for crowdsourced classification. In *ICML*, pages 534–542, 2013.

[18] C. Ho and J. W. Vaughan. Online task assignment in crowdsourcing markets. In *AAAI*, 2012.

[19] J. J. Horton and L. B. Chilton. The labor economics of paid crowdsourcing. In *ACM EC*, pages 209–218, 2010.

[20] N. Kaufmann, T. Schulze, and D. Veit. More than fun and money. worker motivation in crowdsourcing - A study on mechanical turk. In *AMCIS*, 2011.

[21] A. Kittur, J. V. Nickerson, M. S. Bernstein, E. Gerber, A. D. Shaw, J. Zimmerman, M. Lease, and J. Horton. The future of crowd work. In *CSCW*, pages 1301–1318, 2013.

[22] D. B. Martin, B. V. Hanrahan, J. O'Neill, and N. Gupta. Being a turker. In *CSCW*, pages 224–235, 2014.

[23] H. Rahman, S. B. Roy, S. Thirumuruganathan, S. Amer-Yahia, and G. Das. Task assignment optimization in collaborative crowdsourcing. In *IEEE ICDM*, pages 949–954, 2015.

[24] 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.

[25] J. Rogstadius, V. Kostakos, A. Kittur, B. Smus, J. Laredo, and M. Vukovic. An assessment of intrinsic and extrinsic motivation on task performance in crowdsourcing markets. In *ICWSM*, 2011.

[26] S. B. Roy, I. Lykourentzou, S. Thirumuruganathan, S. Amer-Yahia, and G. Das. Task assignment optimization in knowledge-intensive crowdsourcing. *VLDB J.*, 24(4):467–491, 2015.

[27] A. D. Shaw, J. J. Horton, and D. L. Chen. Designing incentives for inexpert human raters. In *CSCW*, pages 275–284, 2011.

[28] T. Wu, L. Chen, P. Hui, C. J. Zhang, and W. Li. Hear the whole story: Towards the diversity of opinion in crowdsourcing markets. *PVLDB*, 8(5):485–496, 2015.

[29] Y. Zheng, J. Wang, G. Li, R. Cheng, and J. Feng. QASCA: A quality-aware task assignment system for crowdsourcing applications. In *SIGMOD*, pages 1031–1046, 2015.

A Probabilistic Framework for Estimating Pairwise Distances Through Crowdsourcing

Habibur Rahman
UT Arlington
habibur.rahman@mavs.uta.edu

Senjuti Basu Roy
NJIT
senjutib@njit.edu

Gautam Das
UT Arlington
gdas@uta.edu

ABSTRACT

Estimating all pairs of distances among a set of objects has wide applicability in various computational problems in databases, machine learning, and statistics. This work presents a *probabilistic framework for estimating all pair distances through crowdsourcing, where the human workers are involved to provide distance between some object pairs.* Since the workers are subject to error, their responses are considered with a probabilistic interpretation. In particular, the framework comprises of three problems : (1) Given multiple feedback on an object pair, how do we combine and aggregate those feedback and create a probability distribution of the distance? (2) Since the number of possible pairs is quadratic in the number of objects, how do we estimate, from the known feedback for a small numbers of object pairs, the unknown distances among all other object pairs? For this problem, we leverage the metric property of distance, in particular, the triangle inequality property in a probabilistic settings. (3) Finally, how do we improve our estimate by soliciting additional feedback from the crowd? For all three problems, we present principled modeling and solutions. We experimentally evaluate our proposed framework by involving multiple real-world and large scale synthetic data, by enlisting workers from a crowdsourcing platform.

1. INTRODUCTION

In this paper we investigate the following problem: *how to obtain pairwise distance values between a given set of objects by using feedback from a crowdsourcing platform?* This problem lies at the core of a plethora of computational problems in databases, machine learning, and statistics, such as top-k query processing, indexing, clustering, and classification problems. We consider an approach where feedback from the crowd is solicited in the form of simple pair-wise comparison questions. As an example, given two images (a, b), workers are asked to rate (in a scale of $[0, 1]$) how dissimilar these two images are. The worker response may be interpreted as the *distance* between the two images. Although the number of pairwise questions is quadratic in the number of objects, the main idea in this paper is to only involve the workers in answering a small number of key pair-wise questions, and to estimate the remaining pair-wise distances using the metric properties of the distance function, in particular the *triangle inequality property* [19] - a property that is true for distance functions that arise in many common applications.

Our iterative crowdsourcing distance estimation framework has three key probabilistic components. When we solicit distance information for a specific object pair from multiple workers, we recognize that due to the subjectivity of the task involved, workers may disagree on their feedback, or may even be uncertain about their own estimate. Thus we develop a probabilistic model for *aggregating* multiple workers feedback to create a single probability distribution of the distance learned about that object pair. Next, given that we have learned the distance distributions of several object pairs from the crowd, we *estimate* the probability distributions of the remaining pairwise distances by leveraging the triangle inequality property of the distances. Finally, if there is still considerable "uncertainty" in the learned/estimated distances and we have an opportunity to solicit additional feedback, we investigate *which object pair* should we choose to solicit the next feedback on. This iterative procedure is continued until all pair-wise distances have been learned/estimated with a desired target certainty (or alternatively, the budget for soliciting feedback from the crowd has been exhausted).

Novelty: There have been a few prior works that have studied computational problems using crowdsourcing that require distance computations. For example, entity resolution [25, 26, 5] problems investigate entity disambiguation, and [22] study top-k and clustering problems in a crowdsourced settings. However, these works have developed their formalism and solutions tightly knit to their specific applications of interest, and do not offer any obvious extension to solve other distance-based applications. For example,the work in [24] is focused on determining whether two objects are the same or not, and not on the broader notion of quantifying the amount of distance between them. In contrast, our proposed framework offers a unified solution to all these computational problems, as they all can leverage our distance estimation framework to obtain the distance between any pair of objects. Please note that once all pair distances are computed, finding the top-k objects, or finding the clusters of the objects is easier to compute. Hence, our problem is more general than the above mentioned body of works. We discuss related work more thoroughly in Section 7.

Challenges and Technical Highlights: There are substantial challenges in formalizing and solving the key prob-

Series ISSN: 2367-2005

10.5441/002/edbt.2017.24

lems that arise in our three probabilistic components. Perhaps the most straightforward is the first component, i.e., how to aggregate the feedbacks received from multiple workers into a single pdf that describes the distance between two objects. There has been several prior works on reconciling the answers from multiple workers, which range from simple majority voting to sophisticated matrix factorization techniques [7, 14] on binary data, or opinion pooling [12, 8, 4] on categorical data. However these methods are largely focused on aggregating Boolean/categorical feedback (e.g., "are these two entities the same?"), whereas in our case we need to merge the potentially diverging and uncertain numeric (distance) feedback from multiple workers into a single probability distribution.

The most challenging aspect of our framework is the second component. Knowing distance distributions of some of the object pairs from the crowd, we have to estimate the probability distributions of the distances of the remaining object pairs, by leveraging the metric property of the distance. While the intuitive idea is simple (e.g., "if a is close to b, and b is close to c, then a and c cannot be too far apart"), the problem is challenging because (a) the known distances themselves are distributions rather than deterministic quantities, and (b) the metric property imposes interdependence between all the pairwise distances in a complex manner. In fact, since there are $n(n-1)/2$ pair-wise distances (where n is the number of objects), each such distance can be assumed to be a random variable such that all distances are jointly distributed in a high dimensional ($n(n-1)/2$) space with interdependencies governed by the triangle inequality. In principle, this joint distribution must be first computed, and then the (marginal) pdfs computed as estimates of the unknown distances. The unknown pairs cannot be estimated in isolation, as a small change in one pdf is likely to disrupt the joint distribution and the triangle inequality property impacting the other pdfs.

We argue that in certain cases, computing the joint distribution may require us to solve a mixture of *over and under-constrained* nonlinear optimization system, whereas in other cases it may reduce to solving an under-constrained system with many feasible solutions ([2]). For the former cases, we formalize the optimization problem as a combination of *least squares and maximum entropy* formulation and present algorithm `LS-MaxEnt-CG` that adopts a conjugate gradient approach [27, 10] to iteratively compute the joint distribution. For the latter cases, the problem reduces to that of maximizing entropy, and we present an algorithm `MaxEnt-IPS` that leverages the idea of *iterative proportional scaling* [23, 21] to efficiently converge to an optimal solution. Both of these solutions, while ideal, only work for small to moderate problem instances since they are exponential in the dimensionality of the joint distribution being estimated. Consequently, we also present a heuristic solution `Tri-Exp` that scales much better and can handle larger problem instances.

In the third component, our task is to decide, from among the remaining unknown object pairs, which one to select for soliciting distance feedback from the crowd. Intuitively, the selected object pair should be the one whose distance (after being learned from the crowd) is likely to reduce the "overall" uncertainty of the remaining unknown distance pdfs the most, i.e., minimize the *aggregated variance* of the remaining pdfs. To solve this problem in a meaningful way, it is critical to be able to model how workers are likely to respond to a solicitation, because their anticipated feedback needs to be taken into account for selecting the most effective pair. Finally, we also recognize that this approach of resolving one object pair at a time by the crowd may be sub-optimal and slow to converge. Thus, we also describe an extension where we "look ahead" and select multiple promising unresolved object pairs, and engage the crowd in simultaneously providing feedback for these pairs.

Summary of Contributions: In summary, we make the following contributions in this paper:

- We consider the novel problem of all-pairs distance estimation via crowdsourcing in a probabilistic settings.
- We identify three key sub-components of our iterative framework, and present formal definitions of problems and the solutions for each of the component (Sections 2,3,4,5).
- We experimentally evaluate our framework using both real world and synthetic datasets to demonstrate its effectiveness (Section 6).

2. DATA MODEL AND PROBLEM FORMULATIONS

We first describe the data model and then formalize the problems considered in this paper.

2.1 Data Model

Objects and Actual Distances: We are given a set $\mathcal{O}$ of n objects, with no two objects being the same. Objects could be images, restaurants, movies, etc. Let $d(i,j)$ be the actual distance between objects i and j. Assume that all distances are normalized within the interval $[0,1]$, where larger values denote larger distances, and that metric properties are satisfied, in particular the *triangle inequality* [19] or *relaxed triangle inequality* [9] property, as we define below. We are interested in using this property for learning all the $\binom{n}{2}$ pairs of distances.

Triangle Inequality Property: For every three objects (i,j,k) that comprise a triangle $\triangle_{i,j,k}$, $d(i,j) \leq d(i,k) + d(k,j)$ and $d(i,j) \geq \left|d(i,k) - d(k,j)\right|$.

To lift the strict notion of triangle inequality, one can consider *relaxed triangle inequality*, that assumes $d(i,j) \leq c.(d(i,k)+d(k,j))$, where c is a known constant that is not too large. Indeed, the *relaxed triangle inequality* [9] property allows us to effectively incorporate subjective human feedback from crowd workers.

Question: A question $Q(i,j)$ to a worker requests feedback on her estimate of $d(i,j)$. The same question Q is directed to m different workers in the available workers pool, in order to gather multiple feedback.

Feedback: Let $f(i,j)$ represents a worker's feedback for the distance. The worker could either give a single value, or a range/distribution of values (if she is uncertain about the distance).[1] Even if the worker gives a single value, if it is known from past history of her performance that this worker is prone to making errors and is only correct with a certain probability p (say, 80%) (referred to as correctness

[1] The latter type of feedback is common in experts opinion aggregation problems [13], where a worker has partial knowledge on a particular topic and her answer reflects that with a distribution over the possible answers.

Notation	Interpretation
$\mathcal{O}$	set of n objects
$d(i,j)$	distance between objects i and j
$Q(i,j)$	asking distance on an object pair (i,j) in $[0-1]$ scale
$f(i,j)$	feedback on object pair (i,j)
$\triangle_{i,j,k}$	triangle formed by objects (i,j,k)
$d^k(i,j), d^u(i,j)$	known and unknown distance between an object pair (i,j), respectively
D^k, D^u	known and unknown set of distances, respectively
$\mathbf{D}$	distance vector
$Pr(\mathbf{D})$	joint probability distribution of $\mathbf{D}$
$\mathbf{W}$	vector representing all buckets of the multi-dimensional histogram of $Pr(\mathbf{D})$
m	m different feedbacks on the same question
$\mathbf{A}$	a Boolean matrix of constraints

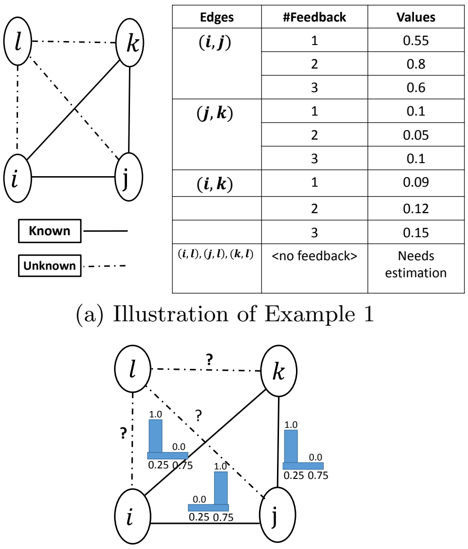
Table 1: Notations

probability), then her single-value feedback can be converted to a more general probability distribution (pdf) over the range $[0,1]$ (e.g.,using techniques described in Section 3). We henceforth assume that the "raw" feedback of the worker has been appropriately processed into a pdf over $[0,1]$.

Known and Unknown Distances: Once a distance question $Q(i,j)$ has been answered by multiple workers, their respective feedbacks needs to be *aggregated* into a single pdf representing how the crowd has estimated the distance between i and j. Exactly how this aggregation is done is the first of the three key challenges of this paper, and is described in detail in Section 3. We denote the random variable described by this pdf as $d^k(i,j)$, where the superscript k denotes that the distance is now "known". Note that it still may not be the *actual* deterministic distance $d(i,j)$, unless the crowd's responses are completely error free, which is often not the case in practice.

Of the $\binom{n}{2}$ distances, let D_k represent the set of known distances, i.e., the ones for which feedback has been obtained from the crowd. Let D_u represent the remaining set of "unknown" distances, i.e., distances between those pair of objects for which feedback has not been explicitly obtained from the crowd. Consider $d^u(i,j) \in D_u$. Even though no information about this distance has yet been solicited, some distributional information about this distance can be derived since it depends on other pairwise distances in a complex manner (due to the triangle inequality property). We discuss this issue next.

Joint Distribution of All Pairs Distances: Consider the set of all distances $D_k \cup D_u$. We may view this set as a *distance vector* $\mathbf{D}$ of length $\binom{n}{2}$, whose every entry is a random variable representing the distance between the respective two objects (i,j). The space of all instances of $\mathbf{D}$ is $[0,1]^{\binom{n}{2}}$, however since the $\binom{n}{2}$ distances are interdependent upon each other due to the metric properties, the *valid instances* are those that satisfy the triangle property, i.e., for the triangle $\triangle_{i,j,k}$ defined by any three objects (i,j,k), the three corresponding distances should satisfy the triangle inequality.



Edges	#Feedback	Values
(i,j)	1	0.55
	2	0.8
	3	0.6
(j,k)	1	0.1
	2	0.05
	3	0.1
(i,k)	1	0.09
	2	0.12
	3	0.15
$(i,l), (j,l), (k,l)$	<no feedback>	Needs estimation

(a) Illustration of Example 1

(b) Distances as Distributions (Histograms with $\rho = 0.5$)

Figure 1: **Illustrative Example.**

Let $Pr(\mathbf{D})$ represent the joint probability distribution of $\mathbf{D}$. Our task is to estimate $Pr(\mathbf{D})$ such that the marginal distribution for a known random variable $d^k(i,j)$ should correspond to the pdf learned from the crowd. We note that once we have an accurate estimation of $Pr(\mathbf{D})$, we can get estimates of the distributions of the unknown random variables $d^u(i,j)$ by computing their marginals. In the next subsection we formalize the problems considered in this paper.

Table 1 summarizes the notations used in the paper.

Example 1. *Image indexing for K-nearest neighbor queries: Our proposed framework is apt to process K-nearest neighbor queries over an image database, where, given a query image, the objective is to obtain an ordered list of images from the database, ordered by how closely they match the query image. To handle such queries faster, one potential avenue is to pre-process the image database and create an index that will cluster the images according to their distance among themselves. Then, as an example, if we have found that a query image $\mathcal{I}$ is far from a database image i and and if the indexes inform us that another image j is close enough to i, then, we may never need to actually compute the distance between $\mathcal{I}$ and j.*

With such an application in mind, consider a toy image database in Figure 1(a) with $n = 4$ images (i,j,k,l), where our eventual goal is to find the distances between all pairs of images. Assume that out of six possible pairs of distances, three are known: (i,j), (j,k), and (i,k). I.e., for each of these pairs, we have solicited feedback from several workers in the crowd, and aggregated the feedbacks to obtain a single probability distribution to describe the distance. The distances of the remaining three pairs are unknown and need to be estimated, again as probability distributions. Furthermore, if we need to solicit further feedback on a question, i.e., get the crowd to provide distance for an unknown pair, we intend to find what is the best question (best pair) to ask.

2.2 Problem Formulations

Recall from Section 1 that the iterative distance estimation framework involves three probabilistic components, which gives rise to three problems that need to be solved: (a) how to aggregate feedbacks from multiple workers for a specific distance question, (b) given some of the learned distances, how to estimate the remaining unknown distances, and (c) which object pair to select next for soliciting feedback from the crowd. In the remainder of this section, we provide formal definitions of these problems, and offer some insights into their complexities.

2.2.1 Problem 1: Aggregation of Workers Feedback for a Specific Object Pair

The first problem may be specified as follows:

PROBLEM 1. *Given a set of m feedbacks for the distance question $Q(i,j)$, where each feedback could be a pdf, aggregate those feedback to create a single pdf for the random variable $d^k(i,j)$.*

Using Example 1, this is akin to aggregating three different feedbacks from three different workers to compute $d^k(i,j)$.

2.2.2 Problem 2: Estimation of Unknown Distances

In this problem we need to leverage the known aggregated distances in D_k to estimate the remaining unknown distances D_u. Obviously, if the distances are completely arbitrary, the unknown distances cannot be computed from the known distances. However, if the distances are *metrics*, in particular satisfying the triangle inequality property, then this property can be leveraged in making better estimates of the unknown distances. Many well known distances are metric, such as, $\ell_2, \ell_1, \ell_\infty$, while other popular distances such as, Jaccard distance and Cosine distance could be transformed to metrics. For us, the challenge is to investigate how this property can be used in the case when the distances are probability distributions rather than fixed deterministic values.

Recall that $\mathbf{D}$ is a random vector representing all the $\binom{n}{2}$ distances, and $Pr(\mathbf{D})$ represents the joint distribution of $\mathbf{D}$. We now describe some important properties that $Pr(\mathbf{D})$ should possess.

The space of all instances of $\mathbf{D}$, i.e., $[0,1]^{\binom{n}{2}}$, may be divided into two as follows: (a) *Valid instances*, i.e., any instance of $\mathbf{D}$ such that all triangles $\triangle_{i,j,k}$ satisfy the triangle inequality, and (b) *Invalid instances*, i.e., any instance of $\mathbf{D}$ such that there exists a triangle $\triangle_{i,j,k}$ that does not satisfy the triangle inequality. Thus $Pr(\mathbf{D})$ should be a function constrained such that the cumulative probability mass over all valid (respectively invalid) instances of $\mathbf{D}$ should be 1 (respectively 0).

Additionally, $Pr(\mathbf{D})$ should be constrained such that the marginal distributions corresponding to the individual random variables in D_k (i.e. the known distances) should agree with the corresponding distance pdfs learned from the crowd. However, this constraint may not be always possible to satisfy, as crowd feedback is inherently an error-prone human activity, which can result in inconsistent feedback that violates the triangle inequality. Thus our task will be to estimate $Pr(\mathbf{D})$ such that the marginal distributions corresponding to individual random variables in D_k are "as close as possible" to the pdfs learned from the crowd.

Once such a $Pr(\mathbf{D})$ has been constructed, the pdfs of the unknown distances can estimated by computing the marginal distributions of each variable in D_u.

In the rest of this subsection, we provide more details of the problem formulation.

Discretization of the pdfs using Histograms: For computational convenience, for the rest of the paper we assume that (single or multi-dimensional) probability distributions are represented as discrete histograms, as is common in databases [17]. In particular, we assume that the $[0,1]$ interval is discretized into equi-width intervals of width ρ (where ρ is a predefined parameter). A r-dimensional pdf is thus represented by a r-dimensional histogram with $(\frac{1}{\rho})^r$ buckets. Each bucket contains a probability mass representing the probability of occurrence of its center value, and the sum of the probabilities of all buckets equals 1.

For the running example in Figure 1(a), we use $\rho = 0.5$. Thus a one-dimensional pdf is represented by a 2-bucket histogram, where the first bucket is between $[0 - 0.5]$ with center at 0.25 and the second bucket is $[0.5 - 1.0]$ with center at 0.75. Figure 1(b) of the running example shows how each known distance (known edge) is represented as a one-dimensional histogram after discretizing and aggregating inputs from multiple users, where the feedback values are replaced by the corresponding bucket centers (we describe details of our techniques for input aggregation, i.e., Problem 1, in Section 3).

Estimating $Pr(\mathbf{D})$: Once we have the histograms for each individual known edge, the joint distribution $Pr(\mathbf{D})$ needs to be estimated as a multi-dimensional histogram with $(\frac{1}{\rho})^{\binom{n}{2}}$ buckets. Our task is to estimate the probability mass of each of these buckets. Using the running example, there are 2^6 buckets, whose centers range from [0.25, 0.25, 0.25, 0.25, 0.25, 0.25] to [0.75, 0.75, 0.75, 0.75, 0.75, 0.75]. Computing the probability mass of a specific bucket, e.g., $Pr(0.25, 0.27, 0.25, 0.25, 0.25, 0.75)$, is equivalent of computing the probability of the simultaneous events $d(i,j) = 0.25$ & $d(j,k) = 0.27$ & $d(i,k) = 0.25$ & $d(i,l) = 0.25$ & $d(k,l) = 0.25$ & $d(j,l) = 0.75$. The computation of $Pr(\mathbf{D})$ can be modeled as a linear system with $(\frac{1}{\rho})^{\binom{n}{2}}$ unknowns, where each unknown represents the probability mass of a bucket. These unknowns have to satisfy three types of linear constraints:

(1) *Constraints imposed by the known pdfs*: $Pr(\mathbf{D})$ should be such that its marginal for any known distance $d^k(i,j)$ should satisfy the corresponding one-dimensional pdf learned from the crowd. Thus, each bucket of each known marginal pdf will generate a linear constraint. In our running example, a one-dimensional bucket such as $Pr(d(i,k) = 0.25)$ will generate a linear equation of the form $\sum Pr(*, *, 0.25, *, *, *) = Pr(d(i,k) = 0.25)$.

(2) *Constraints due to triangle inequality*: Some of the buckets in the joint distribution must have zero probability mass if they violate triangle inequality constraints. In our running example, consider any of the 8 bucket of the form $(0.75, 0.25, 0.25, *, *, *)$. The probability mass of each such bucket has to be set to 0, since $d(i,j) = 0.75$, $d(j,k) = 0.25$ and $d(i,k) = 0.25$ does not satisfy the triangle inequality (this happens irrespective of any combination of the values for the remaining three edges, hence they are represented as '*').

(3) *Probability axiom constraint*: A final constraint requires

that the sum of all the buckets of the joint distribution adds up to 1. In our running example, this implies that $Pr(0.25, 0.25, 0.25, 0.25, 0.25, 0.25) + Pr(0.25, 0.25, 0.25, 0.25, 0.25, 0.75) + \ldots + Pr(0.75, 0.75, 0.75, 0.75, 0.75, 0.75) = 1$.

If $\mathbf{W}$ represents the vector of $(\frac{1}{\rho})^{\binom{n}{2}}$ unknowns, and M represents the set of constraints, then the linear system may be expressed as $\mathbf{AW} = \mathbf{b}$, where $\mathbf{A}$ is a Boolean matrix of size $|M| \times (\frac{1}{\rho})^{\binom{n}{2}}$, and $\mathbf{b}$ is a vector of length $|M|$. Interestingly, as the following discussion shows, solving this linear system is not a straightforward task.

Scenario 1: Over-Constrained Case: In general, an *over-constrained linear system* $\mathbf{AW} = \mathbf{b}$ is one which has no feasible solution [15]. In our case, it is indeed possible that the marginal distributions corresponding to the individual random variables in D_k (i.e. the known distances) that are learned from the crowd gives rise to an over-constrained scenario. This is because crowd feedback is inherently an error-prone human activity, which can result in inconsistent feedback that violates the triangle inequality. For example, $\triangle_{i,j,k}$ in Example 1 has only one deterministic instance with edge weights $d(i,j) = 0.75$, $d(j,k) = 0.25$ and $d(i,k) = 0.25$. Clearly, $\triangle_{i,j,k}$ does not satisfy the triangle inequality, since $d(i,j) > d(i,k) + d(j,k)$. Hence, there is no valid joint distribution $Pr(\mathbf{D})$ which can estimate the known pdfs.In such cases, we estimate $Pr(\mathbf{D})$ such that the marginal distributions corresponding to individual random variables in D_k are "as close as possible" (using *least squares* principle) to the pdfs learned from the crowd. More formally, given $\mathbf{A}$ and $\mathbf{b}$, we estimate $\mathbf{W}$ such that $||\mathbf{AW}-\mathbf{b}||^2$ is *minimized.*

Scenario 2: Under-Constrained Case: In general, an *under-constrained linear system* $\mathbf{AW}$=$\mathbf{b}$ is one which has multiple feasible solutions [15]. In our case, while estimating $\mathbf{W}$, we may also encounter under-constrained scenarios. Using Example 1 and considering triangle $\triangle_{i,j,l}$, we note that any of the following solutions are feasible: $d(i,l) = 0.75$, $d(l,j) = 0.75$, or $d(i,l) = 0.75$, $d(l,j) = 0.25$, or $d(i,l) = 0.25$, $d(l,j) = 0.75$. In such cases, *maximum entropy* principles [23] are used to choose a solution that is consistent with all the constraints but otherwise is as uniform as possible. More formally, the objective is to solve the linear system $\mathbf{AW}$=$\mathbf{b}$ that maximizes the entropy of the joint distribution $-\sum_{w \in \mathbf{W}} Pr(w) \log Pr(w)$.

Scenario 3: Combined Case: Since our problem instances may involve both over and under-constrained scenarios, we unify both into a single minimization problem using a weighted linear combination, where the weight λ can be used to tune the solution to ensure better least square or higher uniformity. Our final problem is described as follows:

Problem 2. *Estimate the joint distribution vector* $\mathbf{W}$ *such that* $f(\mathbf{W}) = \lambda \times ||\mathbf{AW} - \mathbf{b}||^2 + (1-\lambda) \times \sum_{w \in \mathbf{W}} Pr(w) \log Pr(w)$ *is* minimized.

Before we move to our next problem definition, we point out an interesting issue. The exponential size of Problem 2 (the number of buckets in the multi-dimensional histogram is intractably large for most real-world instances) suggests that a complete solution of Problem 2 is prohibitive. Fortunately, we observe that computing the joint distribution is merely a intermediate (and not strictly necessary) objective - our eventual objective is to estimate the one-dimensional pdfs of the unknown distances $d^u(i,j)$. This issue is discussed in more detail in Section 4, and in particular we present heuristics to directly compute the unknown one-dimensional pdfs without having to compute the intermediate joint distribution.

2.2.3 *Problem 3: Asking the Next Best Question*

Recall that our overall approach is an iterative process. If we have the need to solicit further feedback from the crowd, we have to select an object pair from D_u, as human workers have not yet been involved in providing feedback about such pairs. Our objective is to select the most promising pair, i.e., that is most likely to reduce the "uncertainty" of the remaining unknown distances the most. We measure uncertainty by aggregating the *variances* of the remaining unknown distance pdfs (the variance of $d^u(i,j)$ with mean μ is measured as $\sigma^2_{d^u(i,j)} = \sum_{\forall q} p_q * (q-\mu)^2$).

Problem 3. *From the set D_u of the candidate object pairs, choose the next best question $Q(i,j)$ to solicit feedback from the human workers, such that, upon receiving the feedback, the* aggregated variance *over the remaining unknown distances is minimized.*

Aggregated variance, `AggrVar` is formalized in one of the two natural ways, *average variance or largest variance*:
(1) Average variance over the remaining unknown distances:

$$\frac{\sum \sigma^2_{d^u(i',j')}}{|D_u|-1}, d^u(i',j') \in \{D_u - d^u(i,j)\}. \qquad (1)$$

(2) Largest variance over the remaining unknown distances:

$$\max_{d^u(i',j')} \sigma^2_{d^u(i',j')}, d^u(i',j') \in \{D_u - d^u(i,j)\}. \qquad (2)$$

Considering Example 1, this problem will seek to choose the next best question (i.e., edge or object pair) from $D_u = \{(i,l), (j,l), (k,l)\}$.

3. PROBLEM 1: AGGREGATION OF WORKERS FEEDBACK

In this section, we describe our proposed solution `Conv-Inp-Aggr` of aggregating multiple feedbacks on a single object pair (i.e., an edge).

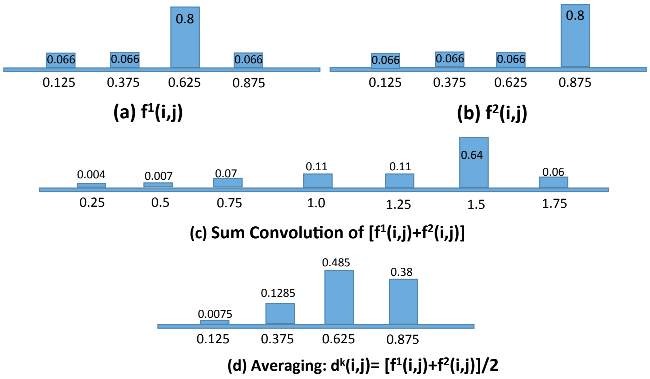


Figure 2: Worker Feedback Aggregation

In general, given a set of m different feedbacks $f^1(i,j), f^2(i,j), \ldots f^m(i,j)$, where each feedback is a random variable describing distance on an object pair (i,j), such that the set of random variables are *independently distributed*, our objective is to define a new random variable whose distribution represents the *average* of the underlying input pdfs, i.e., pdf of $\frac{f^1(i,j)+f^2(i,j)+\ldots+f^m(i,j)}{m}$. The independence assumption allows us to use the prior technique of *sum-convolution* [1] to obtain the sum of the input pdfs and then averaging that convolved pdf to obtain the average.

Algorithm 1 `Conv-Inp-Aggr`

```
1: Input: Set of $m$ feedbacks for $(i, j)$.
2: Perform a sequence of $m - 1$ Sum-convolutions over the feedback pdfs.
3: Calculate $d^k(i, j)$ by re-calibrating the resultant pdf of previous step into pre-specified adjusted range. This step require averaging over the bucket values and reallocate the probability masses accordingly.
4: return $d^k(i, j)$
```

We illustrate this approach using the first two feedbacks for the pair (i, j) in our running example in Figure 1(a). The first worker's feedback (denoted as $f^1(i, j))$ of 0.55 is converted into a pdf. This is shown in Figure 2(a) as a 4-bucket histogram (i.e., with $\rho = 0.25$, buckets with boundaries $[0-0.25]$, $[0.25-0.5]$, $[0.5-0.75]$, $[0.75-1.0]$, and centers at 0.125, 0.375, 0.625, 0.875 respectively). As the feedback value 0.55 is in $[0.5 - 0.75]$, we can assign a probability mass of 1 to this bucket, and 0 to all other buckets. However, if we have prior information that the worker is only correct 80% of the time (correctness probability $p = 0.8$), we can assign a probability mass of 0.8 to the bucket $[0.5 - 0.75]$, and distribute the remaining probability mass uniformly among the remaining three buckets. This latter approach is used to generate the pdf illustrated in Figure 2(a). Similarly, Figure 2(b) shows the pdf for feedback 2 of (i, j).

The sum-convolution of these two pdfs is presented in Figure 2(c). Since the centers of the buckets of each of the individual pdf are between $[0.125, 0.875]$, their sum can be any value between $[0.25, 1.75]$. For each discrete value x between $[0.25, 1.75]$, the probability of $f^1(i, j) + f(^2(i, j)$ equal to x is calculated by computing the joint probability of $f^1(i, j) = x'$ and $f^2(i, j) = x$", such that, $x' + x" = x$.

With $m = 2$ feedbacks, the bucket values are then reassigned to the centers as follows: 0.25 → 0.125, 0.5 → 0.25, ..., 1.75 → 0.875. After this is done, if we have a transformed bucket center with non-zero probability that does not correspond to any of the input buckets, then the mass of that bucket is redistributed to its closest bucket. When two buckets are equally close, the mass is equally divided between the two buckets. As an example, since 1.0 → 0.5 after averaging, but 0.5 does not correspond to any bucket center , the probability mass of $Pr(f^1(i, j) + f^2(i, j) = 1.0)$ gets uniformly split between its two closest centers 0.375 and 0.625. The resultant distribution is given in Figure 2(d).

Figure 1(b) shows the aggregation results for (i, j) of Figure 1(a) with worker being completely accurate ($p = 1.0$) and with $\rho = 0.5$.

Running Time: If each pdf is approximated using an equi-width histogram of width ρ, the time to perform average convolution involving m different pdfs is $O(m \times 1/\rho^2)$ [1].

4. PROBLEM 2: ESTIMATION OF UNKNOWN DISTANCES

In this section, we present our proposed solutions of the problem 2- i.e., how to estimate the distance of the unknown object pairs from the given known distances. Using Example 1, this step is to estimate three unknown distances $D_u = \{(i, l), (j, l), (k, l)\}$, by leveraging the three known distances. We present two alternative solutions - an optimal solution by computing joint distribution that is exponential to the number of object pair $\binom{n}{2}$, and a much faster heuristic alternative.

4.1 Algorithms for Optimal Solution

Recall our proposed formulation in Section 2.2 and note that the optimal solution of computing the unknown distances is to first produce a joint distribution $Pr(\mathbf{D})$ on a high-dimensional space over all $\binom{n}{2}$ object pairs. This is due to our underlying abstraction that assumes that all objects are connected to each other which gives rise to a complete graph - hence the distribution of an unknown edge can not simply be learned in isolation. Once the joint distribution is obtained, the unknown pdfs are to be computed as marginals from the joint distribution. We investigate and design algorithms for the following two scenarios:

(1) As demonstrated in Example 1, our problem can unfortunately be both *over as well as under-constrained.* In fact, when the known pdfs are inconsistent (i.e., do not satisfy triangle inequality), there may not be any *feasible solution* to compute $Pr(\mathbf{D})$ that satisfies all the known pdfs. At the same time, a part of our solution space may still be under-constrained, especially the part that involves the unknown pdfs where multiple feasible solutions may exist.

(2) For the special case when the known pdfs are consistent, the scenario is merely under-constrained and may have multiple feasible solutions, as we describe in Section 4.1.2.

4.1.1 Combined Case

For this scenario, the problem of computing the joint distribution is formalized as an optimization problem (Problem 2) with the objective to minimize a weighted linear combination of least square and negative entropy (notice $-Pr(w) \log Pr(w)$ is the entropy), i.e., $f(\mathbf{W}) = \alpha \times ||\mathbf{AW} - \mathbf{b}||^2 + \beta \times \sum_{w \in \mathbf{W}} Pr(w) \log Pr(w)$ is to be *minimized.* The first part of the formulation is designed for the over-constrained settings, i.e., we satisfy the known pdfs as closely as possible if there is no feasible solution, whereas the second part of the formulation is to handle under-constrained nature of the problem through maximum entropy modeling that will choose the joint distribution model that is consistent with all the constraints but otherwise is as uniform as possible. From the joint distribution $Pr(\mathbf{D})$, we obtain the unknown distance pdfs by computing appropriate marginals.

LEMMA 1. *$f(\mathbf{W})$ is convex.*

PROOF. (Sketch) It can be shown that the linear aggregation of two convex functions is always convex [3], which proves the above lemma. □

Algorithm `LS-MaxEnt-CG`: Based on Lemma 1 $f(\mathbf{W})$ is convex. We propose Algorithm `LS-MaxEnt-CG`, by appropriately adapting nonlinear conjugate gradient algorithms [27, 10] that are popular iterative algorithms to solve such nonlinear convex optimization problems. The overall pseudocode is presented below in Algorithm 2.

Using Example 1 with $\rho = 0.5$, the joint distribution produces the probability for each of the 2^6 buckets that sum up to 1. From this joint distribution, the marginal distributions can be computed for the three unknown edges. (i, l) :

Algorithm 2 LS-MaxEnt-CG

```
1: Input: matrix $A$, constraint vector $b$, vector $\mathbf{W}$ with $\frac{1}{\rho}^{\binom{n}{2}}$ unknown variables, tolerance error $\eta$.
2: Initialize $\mathbf{W}$ with the steepest direction in the first iteration $\Delta\mathbf{W}_0 = -\nabla_{\mathbf{W}} f(\mathbf{W}_0)$
3: In the $i$-th iteration, compute $\beta'_i$ using Fetcher-Reeves method [11].
4: Update the conjugate direction: $s_i = \Delta\mathbf{W}_i + \beta'_i s_{i-1}$.
5: Perform a line search to obtain $\alpha'_i$, $\alpha'_i = \arg\min_{\alpha'} f(\mathbf{W}_i + \alpha' s_i)$.
6: Update the position: $\mathbf{W}_{i+1} = \mathbf{W}_i + \alpha'_i s_i$
7: Repeat Steps $3 - 7$ to until the error $\leq \eta$.
8: return $f(\mathbf{W})$
```

$[0.25 : 0.366, 0.75 : 0.634]$,$(j,l)$: $[0.25 : 0.366, 0.75 : 0.634]$, (k,l) : $[0.25 : 0.366, 0.75 : 0.634]$.

Running Time: It has been shown in [10] that conjugate gradient has a running time complexity of $O(m'\sqrt{\kappa})$, where m' is the number of non-zero entries in the matrix $\mathbf{A}$ and κ is the number of iteration before convergence. However, in our case, as described in Section 2.2, the size of the input matrix $\mathbf{A}$ itself is exponential to the number of object pairs.

4.1.2 Under-Constrained Case

For the under-constrained settings, the optimization function becomes simpler, with the objective to maximize entropy $f(\mathbf{W}) = -\sum_{w \in \mathbf{W}} Pr(w) \log Pr(w)$, while satisfying the known constraints. Each constraint C_i is a restriction on some subset of these possible $(\frac{1}{\rho})^{\binom{n}{2}}$ cells to sum up to some observed value $p(C_i)$. More specifically, each $C_i = \sum(w_i \times I_{i,j})$, where $I_{i,j} = 1$ if j-th cell is included in the constraint C_i, and 0 otherwise.

Algorithm MaxEnt-IPS: It has been shown that the objective function always has a unique solution as long as the constraints are consistent [21]. Of course, this problem can be solved using a general purpose optimization algorithm. However, we propose MaxEnt-IPS, an *iterative proportional scaling (or IPS)* algorithm [23, 21] that exploits the structural property of the objective function and uses the observation that the optimal w_i values can be expressed in the following product form.

$$w_j^{\mu} = \mu_0 \Pi_{C_i} {\mu_i}^{I_{i,j}}$$

For each constraint C_i, there is a constraint μ_i that gets updated inside the IPS algorithm and μ_0 is a normalization constant to ensure that all cells add up to 1. This algorithm iteratively updates the μ_i's and the cell values w_i's. It is guaranteed to converge to the optimal solution as long as all constraints are consistent. Once the histogram buckets W and hence the joint distribution $Pr(\mathbf{D})$ is computed, the unknown marginals are obtained similarly as before. We omit further details and the pseudo-code for brevity but refer to [23, 21] for for more information on the IPS method.

MaxEnt-IPS does not converge for the input presented in Example 1 (b), as it is over-constrained. However, if we modify the example such that the aggregated feedback for (j,k) is 0.75 instead of 0.25, then the following outputs are obtained for the three edges: (i,l) : $[0.25 : 0.333, 0.75 : 0.667]$,$(j,l)$: $[0.25 : 0.333, 0.75 : 0.667]$, (k,l) : $[0.25 : 0.333, 0.75 : 0.667]$.

Running Time: The maximum entropy modeling is known to be NP-hard [18]. The MaxEnt-IPS algorithm terminates based on the convergence of all the μ's. In each iteration it makes updates to all the buckets in the joint distribution, which is exponential in size $(O(\frac{1}{\rho})^{\binom{n}{2}}))$. If MaxEnt-IPS requires κ iterations to converse, the asymptotic complexity of this algorithm is exponential, i.e., $O(\kappa \times (\frac{1}{\rho})^{\binom{n}{2}})$.

4.2 Efficient Heuristic Algorithm

Both the problem variants and their respective solutions studied in Sections 4.1.1 and 4.1.2 first compute the joint distribution over an $\binom{n}{2}$-dimensional space as optimization problems. After that, the unknown distributions are computed from the joint distribution. Even with $n = 5$ objects and $\rho = 0.5$, the joint distribution is to be computed on an $2^{\binom{5}{2}} = 2^{10}$ dimensional space. Due to its exponential nature, computing the joint distribution is practically impossible as n increases. As a realistic alternative, we next present Tri-Exp, an *efficient heuristic algorithm that avoids computing the entire joint distribution, but explores the individual triangles in a greedy manner to estimate the pdfs of the unknown edges.* The pseudo-code is presented in Algorithm 3.

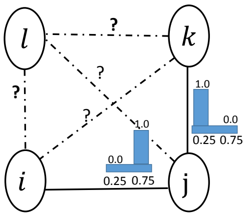


Figure 3: Example to Illustrate Tri-Exp

While Algorithm Tri-Exp avoids computing the joint distribution and instead performs a greedy exploration over the individual triangles one-by-one, there are still considerable challenges - each unknown object pair (edge) is involved in $n-2$ different triangles (with different triangle inequality constraints) and the algorithm must be adapted to estimate the pdf of the unknown edge such that it satisfies all the triangles. In particular, it encounters two scenarios.

Scenario 1: During execution, the algorithm may encounter some triangles which have two edges already known and only the third edge is to be estimated. For such cases, the algorithm *will greedily select that unknown edge that completes the highest number of triangles*, once estimated. When an unknown edge is involved in multiple triangles with two edges known for each triangle, then the final estimated pdf must satisfy the triangle inequality property of all the triangles. We first estimate the pdf of the unknown edge considering each triangle, following which the final pdf is computed by performing the sum-convolution and averaging the convolved pdf (recall Section 3), such that the triangle inequality property is satisfied for all the triangles.

Scenario 2: Another scenario that is likely to occur is when there only exists triangles with two unknown edges. In such cases, both of the unknown edges are jointly estimated, by relying on the known edge.

Solution Considering Scenario 1: As an example, consider Figure 3 and note that based on this greedy selection, at the very first iteration, it will select (i,k) for estimation, as that will complete at least one triangle $\triangle_{i,k,l}$ (because, the two edges of this triangle are already known and the third edge is to be estimated), whereas none of the other unknown edges will complete any triangle. Considering triangle $\triangle_{i,j,k}$, the algorithm will have to apply the triangle inequality property to select the possible ranges of values that (i, j) is allowed to take.

Our method will estimate the pdf of (i, k) as $Pr((i, k) = 0.25) = 0.0$, $Pr((i, k) = 0.75) = 1.0$ considering $\triangle_{i,j,k}$. After that, (i, k) should also be estimated considering another triangle $\triangle_{i,l,k}$. The final pdf of (i, k) must satisfy the triangle inequality property of both of these triangles.

Algorithm 3 `Tri-Exp`: heuristic distance estimation algorithm

```
1: Input: known and unknown distance edges.
2: if There exists triangles with one unknown and two
   known edges then
3:     Greedily select that unknown edge and estimate it
       such that it results in the maximum number of tri-
       angles with all known edges
4: else
5:     When no such triangle is found, consider a triangle
       and estimate two unknown edges jointly
6: end if
7: Perform sum convolution and averaging for all associ-
   ated triangles such that triangle inequality is satisfied
8: Repeat steps $2 - 7$ until all edges are estimated
9: return distance edges
```

Solution Considering Scenario 2: Consider Figure 3 again and assume that (i, k) is estimated in iteration one. Even after that, both $\triangle_{i,j,l}$ and $\triangle_{j,k,l}$ have two unknown edges.

In $\triangle_{j,k,l}$, where both (k, l) and (j, l) are unknowns and are to be estimated using the pdf of the known edge (j, k). Without further knowledge, we calculate the joint distribution for (j, l) and (k, l) by assigning uniform probability to each of these possible values. Once, we get the joint distribution, we calculate the pdfs for both (j, k) and (j, l) which will be exactly equal to each other, which is $\{0.25 : 0.5, 0.75 : 0.5\}$. As before, when multiple triangles are involved with an unknown edge, the pdf of that edge needs to be estimated considering triangle inequality property of all the involved triangles.

`Tri-Exp` outputs the following pdfs for the example in Figure 3 $(i, k) : [0.25 : 0.5, 0.75 : 0.5], (k, l) : [0.25 : 0.61, 0.75 : 0.39], (j, l) : [0.25 : 0.43, 0.75 : 0.57], (i, l) = [0.25 : 0.4, 0.75 : 0.6]$

Running Time: Time complexity of `Tri-Exp` is $O(|D_u|(n\times \frac{1}{\rho}^2 + log(|D_u|))$, where $|D_u|$ is the number of unknown pairs, ρ is the histogram-width, and n is the number of objects. At worst case, $|D_u| = O(n^2)$; in such cases, the algorithm takes cubic time to run. Nevertheless, this analysis shows that the running time of `Tri-Exp` is substantially superior than its exponential counterparts, `LS-MaxEnt-CG` or `MaxEnt-IPS`.

5. PROBLEM 3: ASKING THE NEXT BEST QUESTION

If there is still considerable "uncertainty" in the learned / estimated distances and we have an opportunity to solicit additional feedback, we investigate (in this third problem) which object pair should we choose to solicit the next feedback on. There are several variants of this problem. In the *online* variant, we have the liberty of asking one question at a time and continue the process until all initially unknown pdfs converges "satisfactorily", or a budget B expires. The budget could be used to specify a limit on the number of questions to be asked, or the maximum number of workers to be involved. In the *offline* variant, we need to decide all questions ahead of time so that the fixed budget expired. In the *hybrid* variant, we could solicit workers feedbacks for several batches of say k questions per iteration. In this paper we mainly focus on the online variant, but also present a simple extension to solve the offline problem.

Modeling Possible Worker feedback: Recall the definition of Problem 3 and note that from a given candidate set of questions D_u (where each question is on an object pair), the problem is to select that question which minimizes the aggregated variance `AggrVar` most. The challenge, however, is to be able to *anticipate* possible workers responses that is currently unknown, to be able to guide the optimization problem. A question $Q(i, j) \in D_u$ is essentially a random variable whose distribution has been estimated already by solving Problem 2. Without any further information, the framework has the following limited options to make guesses about future responses of the workers:

(1) The response pdf from the m workers, when aggregated, will be the same as the current estimated pdf of $d^u(i, j)$. Under this scenario, the framework does not learn anything new about $d(i, j)$ and hence `AggrVar` remains unchanged. We therefore do not use this option in our algorithm.

(2) The aggregated response of the worker will be identical to some measures of the current pdf that dictates its central tendency; for example the *mean* μ of the current pdf can used as the anticipated value of the future aggregated feedback.

In this latter case, the pdf of $d^u(i, j)$ changes (its variance becomes 0), and it is also likely to affect the pdfs of other edges (i.e., the joint distribution changes). More intuitively, when a pdf is represented by its mean, the other pdfs (edges) involved with it are likely to demonstrate lower divergence, hence tighter distribution. As described later, this option is used in our algorithm for selecting the next best question.

Consider a very simple example with 3 objects (i, j, k) that satisfy triangle inequality such that $(i, j) : Pr(d(i, j) = 0.125) = 1$; $(i, k) : Pr(d(i, k) = 0.125) = 0.9, Pr(d(i, k) = 0.375) = 0.1$. To satisfy triangle inequality, the pdf of the third edge (j, k) must be between $[0.0, 0.5]$. However, if we substitute (i, k) with its mean 0.15 (considering it as a candidate question), the pdf of (j, k) becomes tighter and only between $[0, 0.275]$. It is easy to notice that the latter pdf of (j, k) will result in a smaller variance in comparison with the former one.

Algorithm `Next-Best-Tri-Exp`: The algorithm for computing the next best question runs in iteration and considers each candidate question $Q(i, j)$ in turn. Then, it considers the impact of changing the current pdf of the object pair to its mean (to emulate workers' feedback). This is done by re-estimating the pdfs in $D_u - d^u(i, j)$. For that, it uses a sub-routine to solve Problem 2, described in Section 4 using any of `LS-MaxEnt-CG`, `MaxEnt-IPS`, or `Tri-Exp` algorithms. Once the unknown pdfs in $\{D_u - d^u(i, j)\}$ are re-estimated, it computes `AggrVar` using either Equation 1 or 2 and maintains the so-far best question by choosing the minimum.

Once all the candidates are evaluated, the best candidate is the one that results in the smallest `AggrVar`. The pseudocode is presented in Algorithm 4. Using Example 1, this returns (i, l) as the next best question, as that minimizes the `AggrVar` based on both formulation of aggregated variance. **Running time:** To choose the next best question, this algorithms has to evaluate each candidate question in D_u. The primary computation time in each candidate question is taken to invoke an algorithm to solve Problem 2 as a subroutine. Therefore, the running time of this algorithm is asymptotically $O(|D_u| \times$ running time of the sub-routine).

Algorithm 4 `Next-Best-Tri-Exp`: Selecting the next best question

```
Input:  known and estimated distance edges.
for $d^u(i,j) \in D_u$ do
    Replace the distribution of $d^u(i,j)$ by its mean
    Select $d^u(i,j) = \text{argmax}_{\forall d^u(i,j) \in D_u}$ AggrVar$(d^u(i,j))$ as the candidate question
end for
return  $d^u(i,j)$
```

Extension to the Offline Problem: If we need to decide how to spend all the budget B ahead of time, we need to decide all the questions offline, we note that the problem becomes computationally more challenging, as there will be an exponential number of possible choices ($\binom{|D_u|}{B}$, assuming the budget allows for B questions) and the ordering of the questions also matters in reducing aggregate variance. However, a simple extension to our current algorithm can effectively solve this offline problem, where we run our online solution B times to select the best B questions greedily. We present experiments on this regard and show that our proposed solution can be effective in solving the offline problem.

6. EXPERIMENTAL EVALUATION

Our development and test environment uses python 2.7 on a Linux Ubuntu 14.04 machine, with Intel core i5 2.3 GHz processor and a 6-GB RAM. All values are calculated as the average of three runs.

6.1 Datasets Description

We use three real world datasets and one synthetic dataset for our experiments. (1) **Image**: The real world dataset is obtained from the PASCAL database[2]. A total of 24 images of 3 different categories are extracted. We generate 3 subsets of size 10, 5, 5 for which we have solicited all pair distance information. Each pair is set up as a HIT (human intelligence task) in Amazon Mechanical Turk (AMT) and we solicit 10 different workers' feedback on the similarity of the images. A total of 50 different workers are involved in this study. (3) **SanFrancisco**: We choose 72 locations from the city of San Francisco and crawl traveling distances (both-ways) among all pair of locations (2556 pairs) using google api[3]. The purpose this dataset is to validate the scalability of our algorithms. Here, we use the traveling distances as worker feedback instead of explicitly soliciting the workers' feedback. (2) **Cora**: This is a real world publication dataset of 1838 records, 190 real world entities. We use this dataset to compare our algorithms with Entity Resolution algorithms in [24]. We choose 3 random instances of this dataset with 20 records, which constitutes of 190 edges. We apply our algorithms in these instances and present our results. (4) **Synthetic**: We generate a large scale synthetic dataset for performing efficiency experiments. Here, we vary from 100 to 400 objects which gives rise from 4950 to 79800 object pairs.Additionally, another small synthetic dataset of 5 objects with 10 edges is generated.

6.2 Implemented Algorithms

(1) Worker Feedback Aggregation: We consider the following algorithms:

(i) `Conv-Inp-Aggr`: This is our proposed convolution based solution to aggregate workers feedback that is described in Section 3.

(ii) `BL-Inp-Aggr`: We implement a baseline algorithm that creates aggregated pdf by calculating the average probability over each discrete bucket center of the input pdfs. Here we ignore the ordinal nature of the feedback scale and treat each bucket as a categorical value.

(2) Estimation of Unknown Edges: We are unaware of any related works that study distance estimation in probabilistic settings.

(i) `Tri-Exp`: This algorithm is described in Section 4.2.

(ii) `LS-MaxEnt-CG`: This algorithm is designed to estimate the unknown edges considering both over and under constrained settings, described in section 4.1.1.

(iii) `MaxEnt-IPS`: This algorithm, described in section 4.1.2, refers to the optimal estimation of unknown edges considering only under-constrained settings.

(iv) `BL-Random`: We design a baseline algorithm that is similar to `Tri-Exp`. It estimates the unknown edges considering triangles; however, unlike `Tri-Exp` (which first attempts to consider the edges that complete the highest number of triangles), `BL-Random` arbitrarily chooses unknown edges and estimates them.

(3) Asking the Next Best Question: These algorithms are designed to demonstrate the effectiveness of the next best question in reducing `AggrVar`, as described in Section 5. As `LS-MaxEnt-CG` and `Maxent-IPS` are computationally prohibitive, we implement `Tri-Exp` and `BL-Random` as subroutines to decide the next best questions. We divide these algorithms into two parts - *Online* and *Offline.*

Online Algorithms: Here we solicit one question at a time to the crowd (i) `Next-Best-Tri-Exp`: This is our proposed solution in Section 5 that uses `Tri-Exp` at each iteration as the subroutine to re-estimate the unknown edges. (ii) `Next-Best-BL-Random`: This is again our proposed solution in Section 5 that uses `BL-Random` at each iteration as the subroutine.

Offline Algorithms: Here we solicit a set of questions ahead of time. (i) `Offline-Tri-Exp`: This is the offline variant of `Next-Best-Tri-Exp` described in Section 5.

(4) Entity Resolution(ER): As discussed in Section 7 on related works, under certain circumstances the problem of *entity resolution*, in particular the techniques proposed in [24], may be considered a special case of the distance estimation problem considered in this paper. Consequently, we experiment with the following algorithms:

(i) `Next-Best-Tri-Exp-ER`: This is a modified vesion of `Next-Best-Tri-Exp` algorithm where we find the number of questions that need to be asked so that `Aggr-Var` is zero.

[2] http://host.robots.ox.ac.uk/pascal/VOC/databases.html

[3] https://developers.google.com/maps/

(ii) `Rand-ER` : We implement the *Random* algorithm from [24]. We call this algorithm `Rand-ER`. This algorithm has a proven complexity of $O(nk)$, where n denotes the number of objects and k denotes the number of clusters/similar entities.

6.3 Experimental Set up

Parameter Settings: Unless otherwise mentioned, we assume $\rho = 0.25$. In other words, there are 4 equi-width buckets with bucket range $[0.0 - 0.25)$, $[0.25 - 0.5)$, $[0.5 - 0.75)$, $[0.75 - 1.0)$ with centers at 0.125, 0.375, 0.625 and 0.875. Depending on the value of p (worker correctness), the distribution of the known edges are created. For example, if a worker provides a feedback of 0.8, with $p = 60\%$, that edge is created by assigning probability of 60% on distance 0.875, and the remaining 40% probability is uniformly assigned to the other buckets. In practice, correctness probability can be obtained by asking a set of screening questions and then by averaging their accuracy. The weight of λ is set to 0.5 (unless otherwise stated) for Problem 2.

Quality Experiments:(i) *Worker Feedback Aggregation*: We use real data for this experiment as this dataset contains multiple workers feedback. We consider each triangle in isolation where all the edge distances are known. Hence, for each edge with 10 different feebacks, we know the ground truth distribution. We use `Conv-Inp-Aggr` and `BL-Inp-Aggr` for aggregating two out of the three edges. Based on our respective algorithm, we estimate the third edge. We then compute the ℓ_2 error of our estimated edge from the ground truth distribution for the third edge. (ii) *Unknown Edge Estimation*: Since `LS-MaxEnt-CG`, `MaxEnt-IPS` are exponential in the number of object pairs (i.e., $S^{{}^nC_2}$), we have to limit our settings to a very small dataset with $n = 5$ nodes and 10 edges. We use the small *Synthetic* dataset, as well as a subset of real world dataset for this experiment. For the *Synthetic* dataset, we consider `MaxEnt-IPS` as the optimal solution, and compare the effectiveness of the other three algorithms by calculating the average ℓ_2 error over the unknown edges, compared to the optimal. Out of the 10 edges, we randomly mark 4 edges as known (and create their distribution as described before), and estimate the remaining 6 unknown edges. For the *Image* dataset, all ground truth distributions are known for the selected 5 objects. Like above, we mark 4 randomly chosen edges to be known and estimate the remaining 6 edges by considering the 4 different algorithms. As before, we present the average ℓ_2 error - but this time in comparison with the ground truth. (iii)*Asking the Next Best Question*: We use the *SanFrancisco* dataset for which we have all pair of ground truth information. At each step, we replace the step of asking a question to the crowd by the ground truth information. The default value of p is 1.0 and the default budget $B = 20$ questions. Number of known edges is is set to 90% of the total edges.

Application to ER: We use *Cora* dataset to perform comparison with ER methods. We assume that each edge is described by a pdf with two ordinal buckets 0 (duplicate) and 1 (not duplicate). We use *number of questions* as our metric which is widely used in ER literature. This value describes the number of questions to be asked before all the entities are resolved. We use 3 random smaller instances of size 20 *Cora* dataset to evaluate our algorithm.

Scalability Experiments: We use the large scale synthetic dataset for the scalability experiments. We vary the following 4 parameters: (i) number of objects n. (ii) number of buckets b' to approximate the pdfs. (iii) number of unknown edges $|D_u|$. (iv) worker correctness p.When one of these aforementioned parameters is varied, the other three are kept constant. The default values for these 4 parameters are, $n = 100$, $|D_u| = 40\%$ of all edges, $b' = 4$, $p = 0.8$. Please note that we primarily present the scalability results for `Tri-Exp` and `BL-Random`, as `LS-MaxEnt-CG` and `MaxEnt-IPS` takes 1.5 days to converge even when $n = 6$.

6.4 Results

6.4.1 Summary of Results

Quality Experiments: Our first experiment on aggregating feedback suggests the superiority of `Conv-Inp-Aggr` over `BL-Inp-Aggr`. For unknown edge estimation, the results indicate that both `Tri-Exp` and `LS-MaxEnt-CG` perform better than the baseline `BL-Random`. For both of them, we observe that with higher worker accuracy (correctness) p, the error increases for all these competing algorithms. While this may appear counter-intuitive, our post-analysis indicates that this is due to the probabilistic nature of our proposed framework and the algorithms, which are most effective, when the workers responses are truly probabilistic. For the third problem, with more questions asked, the `AggrVar` reduces. In both of these aforementioned scenarios, `Next-Best-Tri-Exp` convincingly outperforms `Next-Best-BL-Random`.

Application to ER: Our result demonstrates that `Rand-ER` outperforms `Next-Best-Tri-Exp-ER`. This is expected since our method is designed to solve a more general problem than ER methods - the ER method assumes no worker uncertainty (i.e., workers are always 100% accurate), and it is dependent on the notion of transitive closure, which is a very special case of triangle inequality.

Scalability Experiments: We show that `Tri-Exp` performs reasonably well with the increasing number of objects, buckets, known edges, or worker correctness. The computation time of `BL-Random` is similar to that of `Tri-Exp`, while `Tri-Exp` is qualitatively superior. Therefore, we only present the results of `Tri-Exp` in these experiments. The algorithms that rely on computing joint distribution `LS-MaxEnt-CG`, `MaxEnt-IPS` do not converge beyond a very small number of objects ($n = 5$) even in days.

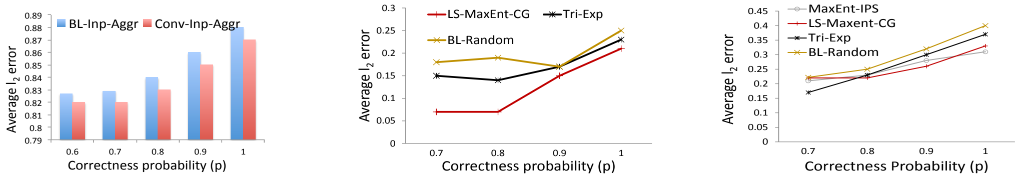
6.4.2 Quality Experiments

(i) *Worker feedback aggregation*: Figure 4(a) shows that `Conv-Inp-Aggr` consistently outperforms the baseline.

(ii) *Estimating Unknown Edges:* We present the results for estimating unknown edges in Figure 4(b) and 4(c). For the synthetic data, `LS-MaxEnt-CG` is superior to the other two methods, while `Tri-Exp` outperforms `BL-Random`. The pattern remains the same for the real data as both `LS-MaxEnt-CG` and `MaxEnt-IPS` exhibit superiority over `BL-Random`. `Tri-Exp` peforms reasonably well for real data. The fact that `LS-MaxEnt-CG` is the best performing algorithm for the real data demonstrates that, in reality, workers may indeed provide inconsistent feedback that do not obey triangle inequality, hence our proposed optimization model is appropriate to capture that settings.

(iii)*Asking the Next Best Question:* We first compare our online algorithms `Next-Best-Tri-Exp` and `Next-Best-BL-Random`.

(a) Varying p: We vary p and present `AggrVar` considering maximum variance. Figure 6(a) presents the results for this

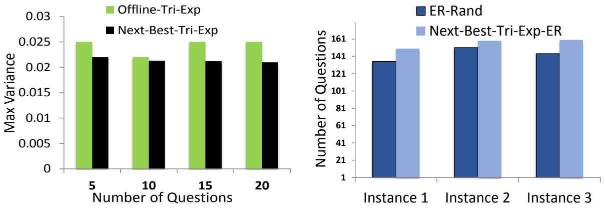


(a) Worker feedback aggregation real data

(b) Error vs p synthetic Data

(c) Error vs p real Data

Figure 4: **Quality Experiments:** i)Worker Feedback Aggregation ii) Unknown Edge Estimation



(a) `Online vs Offline`

(b) `Comparison with ER`

Figure 5: **Experiments for validating Offline algorithms and Entity Resolution**

experiment. While the maximum variance for `Next-Best-BL-Random` and `Next-Best-Tri-Exp` decreases with increasing worker accuracy, latter performs better than the former. For average variance, we encounter the same pattern. Hence, we omit the results for brevity.

(b) Varying B: Our goal here is to test how `AggrVar` reduces with the increasing number of questions (budget B). Figure 6(b) and Figure 6(c) present the outcome of these experiments. It is interesting to observe that with a fairly small number of questions, the `AggrVar` reduces drastically and the system reaches a stable state.

(c) Online vs Offline Experiment: Figure 5(a) presents the result. As expected, `Next-Best-Tri-Exp` performs better than the `Offline-Tri-Exp`, but with very small margin. This result proves that `Offline-Tri-Exp` is very suitable for traditional crowdsourcing framework as online algorithms have high latency.

iv) *Entity Resolution:* Figure 5(b) shows the results for Entity Resolution. Although `Next-Best-Tri-Exp-ER` performs a little worse than `Rand-ER`, we argue that our method is not optimized for finding duplicate entities. Please notice that our method can be applied to find duplicate entities while it is not possible vice versa.

6.4.3 Scalability Experiments

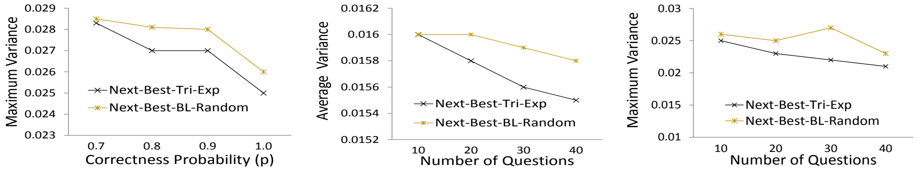
(i) *Worker feedback aggregation:* We observe that the time to aggregate workers feedback is akin to the triangle computation time of `Tri-Exp`. For brevity, we omit the details.
(ii) *Unknown Edge Estimation:* We observe that both heuristic algorithms are equally efficient. Hence, we just present the results of `Tri-Exp`. (a) Varying n: Figure 7(a) presents these results and indicates that `Tri-Exp` converges in a reasonable time, even for higher values of n.
(b) Varying b': Figure 7(b) presents these results and indicates that `Tri-Exp` scales well with increasing b'.
(c) Varying $|D_k|$: Figure 7(c) presents these results and shows that `Tri-Exp` is scalable with increasing number of unknown edges and takes lesser time, as $|D_k|$ increases.
(d) Varying p: Figure 7(d) indicates that the running time of `Tri-Exp` is not affected by p.
(iii)*Asking the Next Best Question:* The running time of `Next-Best-Tri-Exp` and `Next-Best-BL-Random` are similar and dominated by the size of $|D_u|$. These results are similar to that of Figure 7(c) and omitted for brevity.

7. RELATED WORK

User Input Aggregation: Aggregation of opinions is studied in several prior works in AI [12, 8, 4]. An opinion is described as a pdf over a set of categorical values. Since, their methods do not consider the notion of distance, they do not offer an easy extension to our problem. Aggregation of binary feedback(Yes/No) in crowdsourcing is studied in [7, 14]. Their proposed models estimate both worker accuracy and the true answer considering a bipartite graph of workers and tasks. They do not extend beyond binary feedback while we assume a numeric feedback model. [20] study how to find the ranking of a tuple, where tuple scores are given by probability distributions. While this problem is fundamentally different from our first problem, their proposed approach nevertheless justifies our proposed way of convolving multiple pdfs for aggregation.

Distance Estimation: *Distance estimation using crowdsourcing* has gained a significant interest recently for solving a variety of computational problems that require distance estimation, such as top-k, clustering, entity resolution (ER), etc [28, 26, 22]. In most of these works, the dependency on distances is only indirect, as these works are based on asking users to resolve Boolean similarity or ranking questions, e.g., whether two objects are similar or not, or whether one object should be ranked higher than the other. In contrast, our work is the first to directly solicit, from the crowd, the broader notion of numeric distances between objects. In [28], the authors propose a crowdsourced clustering method by leveraging matrix completion techniques, where human workers are involved to annotate objects in a deterministic settings. Entity resolution using crowdsourcing have been studied in [25, 26, 24]. The closest related work is that of [24]. The main differences between this work and ours are: (a) the are only concerned with the Boolean notion of objects equivalency, whereas we try to learn numeric distances between objects, (b) they assume that the crowd can make no mistake, which is unrealistic for distance computations, and (c) they leverage the notion of transitive closure, which is a much simpler notion compared to that of triangle inequality. Therefore their main focus has been on determining the optimal set of questions to ask the crowd, whereas our focus has been on even more basic issues such as how to aggregate uncertain user feedbacks and update the probabilistic distribution models of the distances.

Asking Next Best Question: Our third problem formulation borrows motivation from [16, 26, 6]. [16] describes the problem of finding the maximum item from paiwise comparisons, [26] tackles entity resolution, and [6] studies top-k queries in uncertain database. They all designed algorithms

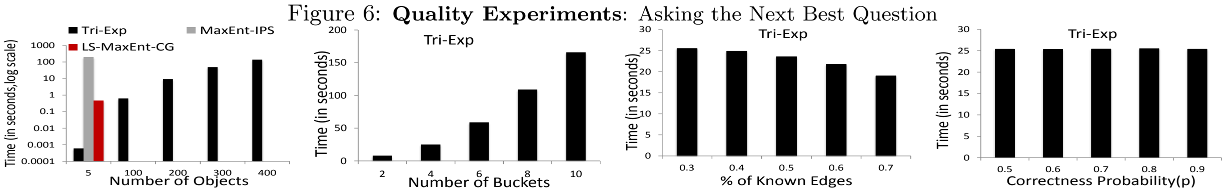


(a) Maximum variance with correctness

(b) Average variance with number of questions

(c) Maximum variance with number of questions

Figure 6: **Quality Experiments**: Asking the Next Best Question



(a) Time vs varying # of objects

(b) Time vs varying # of buckets

(c) Time vs varying # of unknown edges

(d) Time vs varying worker correctness

Figure 7: **Scalability Experiments**: 4 different parameters are varied. Our default settings is $n = 100$, $p = 0.8$, $|D_u| = 50\%$, $b' = 4$.

for finding the next best question which maximize the expected accuracy for their respective problems. Both [16] and [26] prove that finding next best question is NP-Complete. In [6], authors construct a Tree of Possible Ordering(TPO) in order to find the next best question. Although we employ the similar settings, our unique problem formulation requires us to design novel solutions.

8. CONCLUSION

We present a probabilistic distance estimation framework in crowdsourcing platforms that has wide applicability in different domains. One of the novel contributions of the work is to consider worker feedback with probabilistic interpretation and describe the overall framework with three key components.The effectiveness of our proposed solutions are validated empirically using both real and synthetic data.

Acknowledgment

The work of Habibur Rahman and Gautam Das was supported in part by the Army Research Office under grant W911NF-15-1-0020, and a grant from Microsoft Research.

References

[1] B. Arai et al. Anytime measures for top-k algorithms. In *PVLDB*, 2007.

[2] Å. Björck. Numerical methods for least squares problems. *Pressure Rate Deconvolution Methods for Well Test Analysis*, 1996.

[3] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[4] A. Carvalho and K. Larson. A consensual linear opinion pool. *arXiv preprint arXiv:1204.5399*, 2012.

[5] C. Chai, G. Li, J. Li, D. Deng, and J. Feng. Cost-effective crowdsourced entity resolution: A partial-order approach. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 969–984, New York, NY, USA, 2016. ACM.

[6] E. Ciceri, P. Fraternali, D. Martinenghi, and M. Tagliasacchi. Crowdsourcing for top-k query processing over uncertain data. *Knowledge and Data Engineering, IEEE Transactions on*, 28(1):41–53, 2016.

[7] N. Dalvi et al. Aggregating crowdsourced binary ratings. In *WWW*, pages 285–294, 2013.

[8] F. Dietrich and C. List. Probabilistic opinion pooling. 2014.

[9] R. Fagin and L. Stockmeyer. Relaxing the triangle inequality in pattern matching. *International Journal of Computer Vision*, 30:219–231, 1998.

[10] R. Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

[11] R. Fletcher et al. Function minimization by conjugate gradients. *The computer journal*, 1964.

[12] A. Garg, T. Jayram, S. Vaithyanathan, and H. Zhu. Generalized opinion pooling. In *AMAI*, 2004.

[13] D. Gerardi et al. Aggregation of expert opinions. *Games and Economic Behavior*, 2009.

[14] A. Ghosh et al. Who moderates the moderators?: crowdsourcing abuse detection in user-generated content. In *EC*, 2011.

[15] G. H. Golub et al. An analysis of the total least squares problem. *SIAM Journal on Numerical Analysis*, 1980.

[16] S. Guo et al. So who won?: dynamic max discovery with the crowd. In *SIGMOD*, 2012.

[17] Y. Ioannidis. The history of histograms (abridged). In *VLDB*, 2003.

[18] C.-W. Ko et al. An exact algorithm for maximum entropy sampling. *Operations Research*, 1995.

[19] F. W. Lawvere. Metric spaces, generalized logic, and closed categories. *Rendiconti del seminario matématico e fisico di Milano*, 1973.

[20] J. Li and A. Deshpande. Ranking continuous probabilistic datasets. *Proceedings of the VLDB Endowment*, 3(1-2):638–649, 2010.

[21] H. Mannila et al. Prediction with local patterns using cross-entropy. In *KDD*, 1999.

[22] V. Polychronopoulos et al. Human-powered top-k lists. In *WebDB*, 2013.

[23] S. Sarawagi. User-adaptive exploration of multidimensional data. In *VLDB*, 2000.

[24] N. Vesdapunt et al. Crowdsourcing algorithms for entity resolution. *PVLDB*, 2014.

[25] J. Wang et al. Crowder: Crowdsourcing entity resolution. *PVLDB*, 2012.

[26] S. E. Whang et al. Question selection for crowd entity resolution. *PVLDB*, 2013.

[27] C. Xie et al. A maximum entropy-least squares estimator for elastic origin–destination trip matrix estimation. *Transportation Research Part B: Methodological*, 2011.

[28] J. Yi et al. Semi-crowdsourced clustering: Generalizing crowd labeling by robust distance metric learning. In *NIPS*, pages 1772–1780, 2012.

Information Propagation in Interaction Networks

Rohit Kumar
Department of Computer and Decision Engineering
Université Libre de Bruxelles, Belgium
Department of Service and Information System Engineering
Universitat Politécnica de Catalunya, Spain
rohit.kumar@ulb.ac.be

Toon Calders
Department of Computer and Decision Engineering
Université Libre de Bruxelles, Belgium
Department of Mathematics and Computer Science
Universiteit Antwerpen, Belgium
Toon.Calders@uantwerpen.be

ABSTRACT

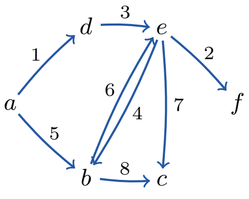
We study the potential flow of information in interaction networks, that is, networks in which the interactions between the nodes are being recorded. The central notion in our study is that of an *information channel.* An information channel is a sequence of interactions between nodes forming a path in the network which respects the time order. As such, an information channel represents a potential way information could have flown in the interaction network. We propose algorithms to estimate information channels of limited time span from every node to other nodes in the network. We present one exact and one more efficient approximate algorithm. Both algorithms are one-pass algorithms. The approximation algorithm is based on an adaptation of the HyperLogLog sketch, which allows easily combining the sketches of individual nodes in order to get estimates of how many unique nodes can be reached from groups of nodes as well. We show how the results of our algorithm can be used to build efficient *influence oracles* for solving the *Influence maximization problem* which deals with finding top k seed nodes such that the information spread from these nodes is maximized. Experiments show that the use of information channels is an interesting data-driven and model-independent way to find top k influential nodes in interaction networks.

Keywords

Influence Maximization, Influence estimation, Information flow mining

1. INTRODUCTION

In this paper, we study information propagation by identifying potential "information channels" based on interactions in a dynamic network. Studying the propagation of information through a network is a fundamental and well-studied problem. Most of the works in this area, however, studied the information propagation problem in static networks or graphs only. Nevertheless, with the recent advancement in data storage and processing, it is becoming increasingly interesting to store and analyze not only the connections in a network but the complete set of interactions as well. In many networks not only the connections between the nodes in the network are important, but also and foremost, how the connected nodes interact with each other. Examples of such networks include email networks, in which not only the fact that two users are connected because they once exchanged emails is important, but also how often and with whom they interact. Another example is that of social networks where people become friends once, but may interact many times afterward, intensify their interactions over time, or completely stop interacting. The static network of interactions does not take these differences into account, even though these interactions are very informative for how information spreads. To illustrate the importance of taking the interactions into account, Kempe et al. [12] showed how the temporal aspects of networks affect the properties of the graph.



(a)

src.	dst.	time
b	c	8
e	c	7
b	e	6
.	..	..
e	f	2
a	d	1

(b)

Figure 1: (a) An example Interaction graph. (b) The interaction in reverse order of time.

Figure 1a gives an example of a toy interaction network. As can be seen, an interaction network is abstracted as a sequence of timestamped edges. A central notion in our study is that of an *information channel*; that is, a path consisting of edges that are increasing in time. For instance, in Figure 1a, there is an information channel from a to e, but not from a to f. This notion of an information channel is not new, and was already studied under the name *time-respecting path* [12] and is a special case of *temporal paths* [26].In contrast to earlier work on information channels we additionally impose a constraint on the total duration of the information channel, thus reflecting the fact that in influence propagation the relevance of the message being propagated may deteriorate over time. To the best of our knowledge, our paper is the first one to study the notion of temporal paths with time constraints in influence propagation on interaction networks.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.25

We propose a method to identify the most influential nodes in the network based on how many other nodes they could potentially reach through an information channel of limited timespan. As such, the information channels form an implicit propagation model learned from data. Most of the related work in the area of information propagation in interaction or dynamic networks uses probabilistic models like the independent cascade(IC) model or the Linear Threshold(LT) model, and tries to learn the influence probabilities that are assumed to be given by these models [13, 4, 3, 6]. Another set of recent work focuses on deriving the hidden diffusion network by studying the cascade information of actions [10, 11] or cascade of infection times [8, 24]. These paper, however, use a very different model of interactions. For example, the work by Goyal et al. [10, 11], every time an activity of a node a is repeated within a certain time span by a node b that is connected to a in the social graph, this is recorded as an interaction. Each user can execute each activity only once, and the strength of influence of one user over the other is expressed as the number of different activities that are repeated. While this model is very natural for certain social network settings, we believe that our model is much more natural for networks in which messages are exchanged, such as for instance email networks because activities such as sending an email can be executed repeatedly and already include the interaction in itself. Furthermore, [11] is not based on information channels, but on the notion of credit-distribution, and [10] does not include the time-respecting constraint for paths.

One of the key differentiators of the techniques introduced here and earlier work is that next to an exact algorithm, we also propose an efficient one-pass algorithm for building an approximate influence oracle that can be used to identify top-k maximal influencers. Our algorithm is based on the same notion as shown in so-called *sliding window HyperLogLog sketch* [15] leading to an efficient, yet approximate solution. Experiments on various interaction networks with our algorithm show the accuracy and scalability of our approximate algorithm, as well as how it outperforms algorithms that only take into account the static graph formed by the connected nodes.

The contribution of this paper are as follows.

- Based on the notion of an *Information Channel*, we introduce the *Influence Reachability Set* of a node in a interaction network.
- We propose an exact but memory inefficient algorithm which calculates the *Influence Reachability Set* of every node in the network in one pass over the list of interactions.
- Next to the exact algorithm, an approximate sketch-based extension is made using a *versioned* HyperLogLog sketch.
- With the influence reachability sets of the nodes in our interaction network, we identify top-k influencers in a model-independent way.
- We propose a new Time Constrained Information Cascade Model for interaction networks derived from the Independent Cascade Model for static networks.
- We present the results of extensive experiments on six real world interaction network datasets and demonstrate the effectiveness of the time window based influence spread maximization over static graph based influence maximization.

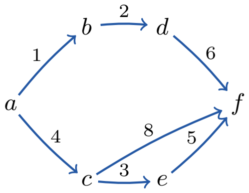


Figure 2: Interaction network example with multiple information channels between node c and f

2. PRELIMINARIES

Let V be a set of nodes. An *interaction* between nodes from V is defined as a triplet (u, v, t), where $u, v \in V$, and t is a natural number representing a time stamp. The interaction (u, v, t) indicates that node u interacted with node v at time t. Interactions are directed and could denote, for instance, the sending of a message. For a directed edge $u \to v$, u is the source node and v is the destination node. An interaction network $G(V, \mathcal{E})$ is a set of nodes V, together with a set $\mathcal{E}$ of interactions. We assume that every interaction has a different time stamp. We will use $n = |V|$ to denote the number of nodes in the interaction network, and $m = |\mathcal{E}|$ to denote the total number of interactions.

Time Constrained Information Cascade Model: For interaction networks, influence models such as the Independent Cascade Model or Linear Threshold Model no longer suffice as they do not take the temporal aspect into account and are meant for static networks. To address this shortcoming, we introduce a new model of Information Cascade for Interaction networks. The *Time Constrained Information Cascade Model* (TCIC) is a variation of the famous *Independent Cascade Model*. This model forms the basis of our comparison with other baselines SKIM [6], PageRank and High Degree. We say a node is *infected* if it is influenced. For a given set of seed nodes we start by infecting the seed nodes at their first interaction in the network and then start to spread influence to their neighbors with a fixed probability. The influence spread is constrained by the time window(ω) specified; i.e, once a seed node is infected at time stamp t it can spread the infection to another node via a temporal path only if the interaction on that path happens between time t and $t + \omega$. For sake of simplicity we use a fixed infection probability in our algorithms to simulate the spread nevertheless node specific probabilities or random probabilities could easily be used as well. In Algorithm 1 we present the algorithm for the TCIC model.

In order to Find highly influential nodes under the TCIC model we introduce the notion of Information Channel.

Definition 1. (Information Channel) *Information Channel ic* between nodes u and v in an interaction network $G(V, \mathcal{E})$, is defined as a series of time increasing interactions from $\mathcal{E}$ satisfying the following conditions: $ic = (u, n_1, t_1), (n_1, n_2, t_2), ...(n_k, v, t_k)$ where $t_1 < t_2 < .. < t_k$. The *duration* of the information channel ic is $dur(ic) := t_k - t_1 + 1$ and the *end time* of the information channel ic is $end(ic) := t_k$. We denote the set of all information channels between u and v as $IC(u, v)$, and the set of all

Algorithm 1 Simulation with a given seed set and window

```
Input: $G(V,E)$ the interaction graph given as a time-ordered list $\ell_G$ of $(u,v,t)$, $\omega$, and $S$ the seed set. $p$ is the probability of infection spread on interaction.
Output: Number of nodes influenced by the seed.
Initially all nodes are inactive and for all activateTime is set to -1.
for all $(u,v,t) \in \ell_G$ do
    if $u \in S$ then
        u.isActive=true
        u.activateTime=t
    end if
    if u.isActive & $(t - u.activateTime) \leq \omega$ then
        With probability $p$
        v.isActive=true
        if u.activateTime > v.activateTime then
            v.activateTime=u.activateTime
        end if
    end if
end for
Return: Count of nodes for which isActive is true.
```

information channels of duration ω or less as $IC_\omega(u,v)$.

Notice that there can exist multiple information channels between two nodes u and v. For example, in Fig 2 there are 2 information channels from a to f. The intuition of the information channel notion is that node u could only have sent information to node v if there exists a time respecting series of interactions connecting these two nodes. Therefore, nodes that can reach many other nodes through information channels are more likely to influence other nodes than nodes that have information channels to only few nodes. This notion is captured by the *influence reachability set*.

Definition 2. (Influence reachability set) The *Influence reachability set (IRS)* $\sigma(u)$ of a node u in a network $G(V,\mathcal{E})$ is defined as the set of all the nodes to which u has an information channel:

$$\sigma(u) \ := \ \{v \in V \mid IC(u,v) \neq \emptyset\} \ .$$

Similarly, the influence set for a given maximal duration ω is defined as

$$\sigma_\omega(u) = \{v \in V \mid \exists ic \in IC(u,v) : dur(ic) \leq \omega\} \ .$$

The *IRS* of a node may change depending on the maximal duration ω. For example, in Figure 2 $\sigma_3(a) = \{b,c,d\}$ and $\sigma_5(a) = \{b,c,d,f\}$. This is quite intuitive because as the maximal duration increases, longer paths become valid, hence increasing the size of the influence reachability set. Once we have the *IRS* for all nodes in a interaction network for a given window we can efficiently answer many interesting queries, such as finding top k influential nodes. Formally, the algorithms we will show in the next section solve the following problem:

Definition 3. (IRS-based Oracle Problem) Given an interaction network $G(V,\mathcal{E})$, and a duration threshold ω, construct a data structure that allows to efficiently answer the following type of queries: *given a set of nodes $V' \subseteq V$, what is the cardinality of the combined influence reachability sets of the nodes in V'; that is:* $\left|\bigcup_{v\in V'} \sigma_\omega(v)\right|$.

First we will present an exact but memory inefficient solution that will maintain the sets $\sigma_\omega(v)$ for all nodes v. Clearly this data structure will allow to get the exact cardinality of the exact influence reachability sets, by taking the unions of the individual influence reachability sets and discarding duplicate elements. The approximate algorithm on it's turn will maintain a much more memory efficient sketch of the sets $\sigma_\omega(v)$ that allows to take unions and estimate cardinalities.

3. SOLUTION FRAMEWORK

In this section, we present an algorithm to compute the IRS for all nodes in an interaction network in one pass over all interactions. In the following all definitions assume that an interaction network $G(V,\mathcal{E})$ and a threshold ω have been given. We furthermore assume that the edges are ordered by time stamp, and will iterate over the interactions in *reverse order* of time stamp. As such, our algorithm is a one-pass algorithm, as it treats every interaction exactly once and, as we will see, the time spent per processed interaction is very low. It is not a streaming algorithm because it can not process interactions as they arrive. The reverse processing order of the edges is essential in our algorithm, because of the following observation.

Lemma 1. Let $G(V,\mathcal{E})$ be an interaction network, and let (u,v,t) be an interaction with a time stamp before any time stamp in $\mathcal{E}$; i.e., for all interactions $(u',v',t') \in \mathcal{E}$, $t' > t$. $G'(V,\mathcal{E} \cup \{(u,v,t)\})$ denotes the interaction network that is obtained by adding interaction (u,v,t) to G. Then, for all $w \in V \setminus \{u\}$, $IRS_\omega(w)$ is equal in G and G'.

Proof. Suppose that $IRS_\omega(w)$ changes by adding (u,v,t) to $\mathcal{E}$. This means that there must exist an information channel ic from w to another node in G' that did not yet exist in G. This information channel hence necessarily contains the interaction (u,v,t). As t was the earliest time in the interaction network G', (u,v,t) has to be the first interaction in this information channel. Therefore w must be u and thus $w \notin V \setminus \{u\}$. □

This straightforward observation logically leads to the strategy of reversely scanning the list of interactions. Every time a new interaction (u,v,t) is added, only the IRS of the source node u needs to be updated. Notice that there is no symmetric definition for the forward scan of a list of interactions; if a new interaction arrives with a time stamp later than any other time stamp in the interaction network, potentially the IRS of every node in the network changes, leading to an unpredictable and potentially unacceptable update time per interaction.

In order to exploit the observation of Lemma 1, we keep a summary of the interactions processed so far.

Definition 4. (IRS Summary) For each pair $u,v \in V$, such that $IC_\omega(u,v) \neq \emptyset$, $\lambda(u,v)$ is defined as the end time of the earliest information channel of length ω or less from u to v. That is:

$$\lambda(u,v) \ := \ \min(\{end(ic) \mid ic \in IC_\omega(u,v)\})$$

The IRS summary $\varphi_\omega(u)$ is now defined as follows:

$$\varphi_\omega(u) = \{(v, \lambda(u,v)) \mid v \in IRS_\omega(u)\} \ .$$

That is, we will be keeping for every node u the list of all other nodes that are reachable by an information channel of duration at most ω. Furthermore, for every such reachable node v, we keep the earliest time it can be reached from u by an information channel. The IRS of a node u can easily be computed from $\varphi_\omega(u)$ as $\sigma_\omega(u) = \{v \mid \exists t : (v,t) \in \varphi(u)\}$. On the other hand, the information stored in the summary consisting of $\varphi(u)$ for every u is sufficient to efficiently update it whenever we process the next edge in the reverse order as we shall see.

Example 1. In Figure 2, $\varphi_3(a) = \{(b,1),(d,2),(c,4)\}$ and $\varphi_3(c) = \{(f,5),(e,3)\}$. There are 2 information channels between c and f, one with $dur(ic) = 1$ and $end(ic) = 8$ and another with $dur(ic) = 3$ and $end(ic) = 5$ and hence $\lambda(c,f) = 5$.

3.1 The Exact algorithm

We illustrate our algorithm using the running example in Figure 1a. Table 1b shows all the interactions for the graph reverse ordered by time stamp. Recall that we process the edges in time decreasing order. The algorithm is detailed in Algorithm 2. First, we initialize all $\varphi(u)$ to the empty set. Then, whenever we process an interaction (u,v,t), we know from Lemma 1 that only the summary $\varphi(u)$ may change. The following lemma explains how the summary $\varphi(u)$ changes:

Lemma 2. Let $G(V,\mathcal{E})$ be an interaction network, and let (u,v,t) be an interaction with a time stamp before any time stamp in $\mathcal{E}$; i.e., for all interactions $(u',v',t') \in \mathcal{E}$, $t' > t$. $G'(V,\mathcal{E} \cup \{(u,v,t)\})$ denotes the interaction network that is obtained by adding the interaction (u,v,t) to G. Let $\varphi'(u)$ denote the summary of u in G' and $\varphi(u)$ that in G. Then, $\varphi'(u) = \downarrow(\{(v,t)\} \cup \varphi(u) \cup \{(z,t') \in \varphi(v) \mid t' - t + 1 \leq \omega\})$, where $\downarrow(A)$ denotes $A \setminus \{(v,t) \in A \mid \exists (v,t') \in A : t' < t\}$.

Proof. Let ic be an information channel of duration maximally ω from u to z in G' that minimizes $end(ic)$. Then there are three options: (1) ic is the information channel from u to v formed by the single interaction (u,v,t) that was added. The end time of this information channel is t. (2) ic was already present in G, and hence $(z, end(ic)) \in \varphi(u)$, or (3) ic is a new information channel. Using similar arguments as in the proof of Lemma 1, we can show that ic needs to start with the new interaction and that the remainder of ic forms an information channel ic' from v to z in G with $end(ic') = end(ic)$. In that case $(z, end(ic)) \in \varphi(v)$. Given the constraint on duration we furthermore need to have $end(ic) - t + 1 \leq \omega$. Hence, $\varphi'(u)$ needs to be a subset of $\{(v,t)\} \cup \varphi(u) \cup \{(z,t') \in \varphi(v) \mid t' - t + 1 \leq \omega\}$, and we can obtain $\varphi'(u)$ by only keeping those pairs that are not dominated. □

Example 2. Figure 1a represents a small interaction network and Table 1b shows the edges in order of time. For $\omega = 3$ the Influence Summary Set will update as follows:

		a	b	c	d	e	f
	φ	{}	{}	{}	{}	{}	{}

		a	b	c	d	e	f
$\xrightarrow{(b,c,8)}$	φ	{}	**(c, 8)**	{}	{}	{}	{}

		a	b	c	d	e	f
$\xrightarrow{(e,c,7)}$	φ	{}	$(c,8)$	{}	{}	**(c, 7)**	{}

		a	b	c	d	e	f
$\xrightarrow{(b,e,6)}$	φ	{}	**(c, 7)(e, 6)**	{}	{}	$(c,7)$	{}

		a	b	c	d	e	f
$\xrightarrow{(a,b,5)}$	φ	**(b, 5)** **(c, 7)** **(e, 6)**	(c,7) (e,6)	{}	{}	$(c,7)$	{}

		a	b	c	d	e	f
$\xrightarrow{(e,b,4)}$	φ	(b,5) (c,7) (e,6)	(c,7) (e,6)	{}	{}	(c,7) **(b, 4)**	{}

		a	b	c	d	e	f
$\xrightarrow{(d,e,3)}$	φ	(b,5) (c,7) (e,6)	(c,7) (e,6)	{}	**(e, 3)** **(b, 4)**	(c,7) (b,4)	{}

		a	b	c	d	e	f
$\xrightarrow{(e,f,2)}$	φ	(b,5) (c,7) (e,6)	(c,7) (e,6)	{}	(e,3) (b,4)	(c,7) (b,4) **(f, 2)**	{}

		a	b	c	d	e	f
$\xrightarrow{(a,d,1)}$	φ	(b,5) (c,7) **(e, 3)** **(d, 1)**	(c,7) (e,6)	{}	(e,3) (b,4)	(c,7) (b,4) (f,2)	{}

While processing the edge $(b,e,6)$, first we add $(e,6)$ in the summary of d and then add $(c,7)$ from the summary of e in summary of b. As the summary of b already had $(c,8)$, the value will be updated. Next, during the processing of edge $(a,b,5)$ the summary of a is updated first by adding $(b,5)$ then while merging the summary of b in a we will ignore $(e,8)$ because the duration of the channel is 4 and the permitted window length is 3. The only addition is hence $(c,7)$.

Theorem 1. *Algorithm 2 updates the IRS summary correctly.*

Proof. This proof follows by induction. For the empty list of transactions, the algorithm produced the empty summary. This is our base case. Then, for every interaction that is added in the for loop, it follows from Lemma 1 and Lemma 2 that the summaries are correctly updated to form the summary of the interaction graph with one more (earlier) interaction. After all interactions have been processed, the summary is hence that of the complete interaction graph. □

Lemma 3. Algorithm 2 runs in time $\mathcal{O}(mn)$ and space $\mathcal{O}(n^2)$, where $n = |V|$ and $m = |\mathcal{E}|$.

Proof. Each edge in $\mathcal{E}$ is processed exactly once and for each edge, both Add and Merge are called once. We assume that the summary sets $\varphi(u)$ are implemented with hash tables such that looking up the element (v,t) for a given v takes constant time only. Under this assumption, the Add function has constant complexity. The Merge function calls Add for every item in $\varphi(v)$ at least once. The number of items in $\varphi(v)$ is upper bounded by n and hence the time complexity of one merge operation is at most $\mathcal{O}(n)$. This leads to the upper bound $\mathcal{O}(mn)$ in total.

For the space complexity, note that in the worst case for each node there is an information channel to every other node of duration at most ω. In that case, the size of the individual summary $\varphi(v)$ of every node v is $\mathcal{O}(n)$ which leads to a space complexity of $\mathcal{O}(n^2)$ in total. □

As we can see from Lemma 3 the memory requirements for the exact algorithm is in worst case quadratic in the

Algorithm 2 Influence set with Exact algorithm

```
Input: Interaction graph $G(V,\mathcal{E})$. $\ell_G$ is the list of interactions reversely ordered by time stamp
    Threshold $\omega$ (maximum allowed duration of an influence channel)
Output: $\varphi(u)$ for all $u \in V$

    function ADD($\varphi(u)$,$(v,t)$)
        if $\exists t^{'} : (v,t^{'}) \in \varphi(u)$ then
                                        ▷ There is at most one such entry
            if $t < t^{'}$ then
                $\varphi(u) = (\varphi(u) \setminus (v,t^{'})) \cup (v,t)$
            end if
        else
            $\varphi(u) = \varphi(u) \cup \{(v,t)\}$
        end if
    end function

    function MERGE($\varphi(u)$,$\varphi(v)$,$t$,$\omega$)
        for all $(x,t_x) \in \varphi(v)$ do
            if $t_x - t < \omega$ then ADD($\varphi(u)$,$(x,t_x)$)
            end if
        end for
    end function

    Initialize: $\varphi(u) \leftarrow \emptyset \quad \forall u \in V$
    for all $(u,v,t) \in \ell_G$ do
        ADD($\varphi(u)$,$(v,t)$)
        MERGE($\varphi(u)$,$\varphi(v)$,$t$,$\omega$)
    end for
```

number of nodes of the graph. This will not scale well for large graphs as we want to keep this data structure in memory for efficient querying. Hence in the next section we will present an approximate but more memory and time efficient version of the algorithm.

3.2 Approximate Algorithm

Algorithm presented in the previous section computes the *IRS* exactly, albeit at the cost of high space complexity and update time. In this section, we describe an approximate algorithm which is much more efficient in terms of memory requirements and update time. The approximate algorithm is based on an adaptation of the HyperLogLog sketch [9].

3.2.1 HyperLogLog Sketch

A HyperLogLog (HLL) sketch [9] is a probabilistic data structure for approximately counting the number of distinct items in a stream. Any exact solution for counting the number of distinct items in a stream would require $\mathcal{O}(N)$ space with N the cardinality of the set. The HLL sketch, however, approximates this cardinality with no more than $\mathcal{O}(\log(\log(N)))$ bits. The HLL sketch is an array with $\beta = 2^k$ cells $(c_1, \ldots, c_\beta)$, where k is a constant that controls the accuracy of the approximation. Initially all cells are 0. Every time an item x in the stream arrives, the HLL sketch is updated as follows: the item x is hashed deterministically to a positive number $h(x)$. The first k bits of this number determines the 0-based index of the cell in the HLL sketch that will be updated. We denote this number $\iota(x)$. For the remaining bits in $h(x)$, the position of the least significant bit that is 1 is computed. This number is denoted $\rho(x)$. If $\rho(x)$ is larger than $c_{\iota(x)}$, $c_{\iota(x)}$ will be overwritten with $\rho(x)$.

For example, suppose that we use a HLL sketch with $\beta = 2^2 = 4$ cells. Initially the sketch is empty:

0	0	0	0

Suppose now item a arrives with $h(a) = 1110100110010110_b$. The first 2 bits are used to determine $\iota(a) = 11_\beta = 3$. The rightmost 1 in the binary representation of $h(a)$ is in position 2, and hence c_3 becomes 2. Suppose that next items arrive in the stream with $(c_{\iota(x)}, \rho(x))$ equal to: $(c_1, 3)$, $(c_0, 7)$, $(c_2, 2)$, and $(c_1, 2)$, then the content of the sketch becomes:

7	3	2	2

It is clear that duplicate items will not change the summary. Furthermore, for a random element x, $P(\rho(x) \geq \ell) = 2^{-\ell}$. Hence, if d different items have been hashed into cell c_ι, then $P(c_\iota \geq \ell) = 1 - (1 - 2^{-\ell})^d$. This probability depends on d, and all c_i are independent. Based on a clever exploitation of these observations, Flajolet et al. [9] showed how the number of distinct items in a stream can be approximated from the HLL sketch. Last but not least, two HLL sketches can easily be combined into a single sketch by taking for each index the maximum of the values in that index of both sketches.

3.2.2 Versioned HLL Sketch

The HLL sketch is an excellent tool for our purpose; every time an edge (a, b) needs to be processed (recall that we process the edges in reverse chronological order), all nodes reachable by an information channel from b, are also reachable by an information channel from a. Therefore, if we keep the list of reachable nodes as a HLL sketch, we can update the reachable nodes from a by unioning in the HLL sketch of the reachable nodes from b into the HLL sketch of those reachable from a. One aspect, however, that is not taken into account here is that we only consider information channels of length ω. Hence, only those nodes reachable from b by an information channel that ends within time window ω should be considered. Therefore, we developed a so-called *versioned* HLL sketch vHLL. The vHLL maintains for each cell c_i of the HLL a list L_i of $\rho(x)$-values together with a timestamp and is updated as follows: let $t_{current}$ be the current time; periodically entries (r, t) with $t - t_{current} + 1 > \omega$ are removed from vHLL. Whenever an item x arrives, $\rho(x)$ and $\iota(x)$ are computed, and the pair $(\rho(x), t_{current})$ is added to the list $L_{\iota(x)}$. Furthermore, all pairs (r, t) such that $r \leq \rho(x)$ are removed from $L_{\iota(x)}$. The rationale behind the update procedure is as follows: at any point in time $t_{current}$ we need to be able to estimate the number of elements x that arrived within the time interval $[t_{current}, t_{current} + \omega - 1]$. Therefore it is essential to know the maximal $\rho(x)$ of all x that arrived within this interval. We keep those pairs (r, t) in L_ι such that r may, at some point, become the maximal value as we shift the window further back in time. It is easy to see that any pair (r, t) such that $r \leq \rho(x)$ for a newly arrived x at $t_{current}$ will always be dominated by $(\rho(x), t_{current})$. On the other hand, if $\rho(x) < r$ we still do have to store $(\rho(x), t_{current})$ as (r, t) will leave the window before $(\rho(x), t_{current})$ will.

Example 3. Suppose that the elements e, d, c, a, b, a have to be added to the vHLL. Recall that we process the stream in reverse order, hence the updates are processed in the following order: (a, t_6), (b, t_5), (a, t_4), (c, t_3), (d, t_2),

(e, t_1). Let ι and ρ be as follows for the elements in V:

item	a	b	c	d	e
ι	1	3	3	2	2
ρ	3	1	2	2	1

The subsequent vHLL sketches are respectively the following:

$$\begin{array}{|c|c|c|c|}\hline \{\} & \{\} & \{\} & \{\} \\ \hline\end{array}$$

$$\xrightarrow{(a,t_6)} \begin{array}{|c|c|c|c|}\hline \{\} & (3,t_6) & \{\} & \{\} \\ \hline\end{array}$$

$$\xrightarrow{(b,t_5)} \begin{array}{|c|c|c|c|}\hline \{\} & (3,t_6) & \{\} & (1,t_5) \\ \hline\end{array}$$

$$\xrightarrow{(a,t_4)} \begin{array}{|c|c|c|c|}\hline \{\} & (3,t_4) & \{\} & (1,t_5) \\ \hline\end{array}$$

$$\xrightarrow{(c,t_3)} \begin{array}{|c|c|c|c|}\hline \{\} & (3,t_4) & \{\} & (2,t_3) \\ \hline\end{array}$$

$$\xrightarrow{(d,t_2)} \begin{array}{|c|c|c|c|}\hline \{\} & (3,t_4) & (2,t_2) & (2,t_3) \\ \hline\end{array}$$

$$\xrightarrow{(e,t_1)} \begin{array}{|c|c|c|c|}\hline \{\} & (3,t_4) & (2,t_2),(1,t_1) & (2,t_3) \\ \hline\end{array}$$

Notice that also two vHLL sketches can be easily combined by merging them. For each cell ι, we take the union of the respective lists L_ι and L'_ι and remove all pairs (r, t) in the result that are dominated by a pair (r', t') that came from the other list with $t' < t$ and $r' \geq r$. If the lists are stored in order of time, this merge operation can be executed in time linear in the length of the lists.

Example 4. Consider the following two vHLL sketches:

$$\begin{array}{|c|c|c|c|}\hline \{\} & (3,t_4) & (1,t_1),(2,t_2) & (2,t_3) \\ \hline\end{array}$$

$$\begin{array}{|c|c|c|c|}\hline \{(5,t_1)\} & (3,t_2) & (4,t_3) & (1,t_4) \\ \hline\end{array}$$

The result of merging them is:

$$\begin{array}{|c|c|c|c|}\hline \{(5,t_1)\} & (3,t_2) & (1,t_1),(2,t_2),(4,t_3) & (2,t_3) \\ \hline\end{array}$$

Note that adding versioning to the HLL sketch comes at a price.

Lemma 4. The expected space for storing a vHLL sketch for a window length ω is $\mathcal{O}(\beta(\log(\omega)^2))$.

Proof. The size of each pair (r, t) stored in a list L_ι is dominated by t and takes space $\mathcal{O}(\log(\omega))$. In worst case, all elements in the window $x_{current}, \ldots, x_{current+\omega-1}$ are different and all arrive into the same cell c_ι. In that case, the expected number of pairs in $L\iota$ is $E[X_1 + X_2 + \ldots + X_{\omega-1}]$ where X_i denotes the following statistical variable: X_i equals 1 if $(\rho(x_i), t_{current+i-1})$ is in $L\iota$ and 0 otherwise. This means that $X_i = 1$ if and only if $\rho(x_i) > \max\{\rho(x_1), \ldots, \rho(x_{i-1})\}$. As each $\rho(x_j)$, $j \leq i$ has the same chance to be the largest, $P(X_i = 1) \leq \frac{1}{i}$. Hence we get:

$$E[|L_\iota|] \leq E[X_1 + \ldots + X_{\omega-1}] \leq \sum_{i=1}^{\omega} \frac{1}{i} = \mathcal{O}(\log(\omega)) \ .$$

□

3.2.3 vHLL-Based Algorithm

The approximate algorithm is very similar to the exact algorithm 2; instead of using exact sets we use the more compact versioned HyperLogLog sketch. Add and Merge are the only functions which need to be updated as per the new sketch everything else will remain the same as shown in algorithm 2. We will just present the ApproxAdd and ApproxMerge functions in Algorithm 3.

Lemma 5. The expected time complexity for Algorithm 3 is $\mathcal{O}(m\beta(\log(\omega))^2)$, where $n = |V|$ and $m = |\mathcal{E}|$.

Algorithm 3 Approximate Algorithm for IRS

```
function ApproxAdd($\varphi(u)$,$(\rho(v), t)$,$\iota(v)$)
    if $\exists(\rho, t') \in L_\iota : (\rho, t')$ dominates $(\rho(v), t)$ then
        Ignore $(\rho(v), t)$
    else
        if $\exists(\rho, t') \in L_\iota : (\rho(v), t)$ dominates $(\rho, t')$ then
            remove $(\rho, t')$ from $L_\iota$
        end if
        Append $(\rho(v), t)$ in $L_\iota$
    end if
end function
function ApproxMerge($\varphi(u)$,$\varphi(v)$,$t$,$\omega$)
    while $i < \beta$ do
        for all $(x, t_x) \in L_i$ do            ▷ Iterate over $\varphi(v)$
            if $t_x - t < \omega$ then
                ApproxAdd($\varphi(u)$,$(x, t_x)$, $i$)
            end if
        end for
        $i + +$
    end while
end function
```

Proof. In the ApproxMerge function the while loop will run for β iterations and the inner for loop will run for an expected of $\log(\omega)$ items(from Lemma 4). Hence time complexity would be $\mathcal{O}(\beta \log(\omega)\mathcal{O}(\text{ApproxAdd}))$.

Now in the ApproxAdd function there are at-most $\log(\omega)$ comparisons, hence $\mathcal{O}(\text{ApproxAdd}) = \mathcal{O}(\log(\omega))$. For each edge ApproxAdd and ApproxMerge are called only once. Hence $\mathcal{O}(m\beta(\log(\omega))^2)$ is the expected time complexity. □

Lemma 6. The expected space complexity for the Algorithm 3 is $\mathcal{O}(n\beta(\log(\omega))^2)$, where $n = |V|$ and $m = |\mathcal{E}|$.

Proof. From Lemma 4 the expected size of one vHLL sketch is $\mathcal{O}(\beta(\log(\omega))^2)$. There will be only one vHLL sketch for each node, hence, expected space complexity is $\mathcal{O}(n\beta(\log(\omega))^2)$. □

4. APPLICATIONS

4.1 Influence Oracle:

Given the *Influence Reachability Set* of an interaction network computing the influence spread of a given seed set, $S \subseteq V$ is straightforward. The influence spread for seed set S is computed as:

$$Inf(S) = \bigcup_{u \in S} \sigma(u) \qquad (1)$$

HyperLogLog sketch union requires taking the maximum at each bucket index ι which is very efficient, so the the time complexity would be $\mathcal{O}(|S|\ell)$.

4.2 Influence Maximization:

Influence Maximization deals with the problem of finding top k seed nodes which will maximize the influence spread. After the pre processing stage of computing *IRS* we can use a greedy approach to find the top-k seed nodes by using the Influence oracle. First we show the complexity of the top-k most influential nodes problem is NP-hard and then show that the Influence oracle function is monotone and submodular. Hence we can use a greedy approximation approach.

Lemma 7. Influence maximization under the *Influence Reachability Set* model is NP-hard.

Proof. Given the *Influence Reachability Set* for all the nodes the problem of finding a subset of k nodes such that the union is maximum is a problem which is similar to the problem of maximum coverage problem. As the later is a NP-hard problem we deduce that the given problem is NP-hard. □

Lemma 8. The influence function $\sigma(S)$ is submodular and monotone.

Proof. First we will prove that $Inf(S)$ is a submodular function. Let S and T be two sets of seed nodes such that $S \subset T$. Let x be another node not in T. Now, let the marginal gain of adding x in S, i.e., $Inf(S+x) - Inf(S) = P$. P is the set of those nodes for which there is no path from S and hence these should belong to $Inf(x)$. Let the marginal gain of adding x in T,i.e., $Inf(T+x) - Inf(T) = P'$. It is clear that $P' \subseteq P$, as otherwise there will be a node u for which there is a path from S but not from T and this is not possible given $S \subset T$. Hence $Inf(S+x) - Inf(S) \geq Inf(T+x) - Inf(T)$.

It is obvious to see the that *Inf* is monotone as it is a increasing function, adding a new node in the seed set will never decrease the influence, and hence if $S \subset T$ then $Inf(S) \leq Inf(T)$ □

Greedy Approach for Influence Maximization:

Algorithm 4 outlines the details for the greedy approach. We start by first sorting the nodes based on the size of the *Influence Reachability Set*. The node with maximum IRS set size becomes the most influential node and is taken as the first node in seed set. Next at each stage we iterate through the sorted list and check the gain by using influence oracle of the already selected nodes and the new node. The node which results in maximum gain is added into the seed set.

Algorithm 4 Influence Maximization using IRS

```
Input: The Influence set $\sigma_u \forall u \in V$ and the number of
    seed nodes to find is k
    initialize $selected \leftarrow \emptyset \wedge covered \leftarrow \emptyset$
    $Sort$ $u \in V$ descending with respect to $|\sigma_u|$. Save this
    sorted list as $\ell$
    while $selected < k$ do
        $gain = 0$ ; $u_s = \emptyset$
        for all $u \in \ell$ do
            if $|covered \cup \sigma_u| - |covered| > gain$ then
                $gain = |covered \cup \sigma_u| - |covered|$
                $u_s = \{u\}$
            end if
            if $gain > \sigma_u$ then
                break;
            end if
        end for
        $selected \leftarrow selected \cup u_s$; $covered \leftarrow covered \cup \sigma_{u_s}$
    end while
```

5. RELATED WORK

The problem of Influence Maximization and Influence spread prediction is a well know problem. Broadly, the work in this area can be categorized into two main categories. The first category is based on static graphs [7, 23, 13, 6] where the underlying graph is already given and the probability of a node getting influenced is derived from probabilistic simulations. The second category is data driven, where the underlying influence graph is derived based on a relationship such as friendship between two users or common action within a specified time [24, 8, 11, 10]. The static graph approaches do not capture the dynamics of real networks such as social media and hence the data driven approaches are more suitable.

Static graph.

The Influence Maximization problem in social network was first studied by Richardson et al. [7, 23] where they formalized the problem with a probabilistic model. Later Kempe et al. [13] proposed a solution using discrete optimization. They proved that the Influence Maximization problem is NP-hard and provided a greedy algorithm to select seed sets using maximum marginal gain. As the model is based on Monte Carlo simulations, it is not scalable for large graphs. Later improvements were proposed by Chen et al. [4] using the DegreeDiscountand *prefix excluding maximum influence in-arborescence* (PMIA) [3] algorithms. Both algorithms are heuristic-based. Leskovec et al. proposed the Cost-Effective Lazy Forward (CELF) [17] mechanism to reduce the number of simulations required to select seeds. All of the above-mentioned studies focus on static graph and do not take the temporal nature of the interactions between different nodes into consideration. The latest work on the static graph Influence Maximization problem by Cohen et al. [6] is the fastest we have come across which scales to very large graphs. We compare our seed sets and their influence spread with the seeds selected by their algorithm SKIM. Related work on information flow mining on static graph may be found in [14, 17, 19, 22, 21]. Lie et al. in [20] and Chen et al. in [2] independently proposed the first time constrained Influence Maximization solutions for static graph. Their work considers the concept of time delay in information flow. They assign this delay at individual node level based on different probabilistic models and not the information channels or pathways between the nodes.

Data Driven approach.

There are a few recent work which consider the temporal aspect of the graph and are based on real interaction data. Goyal et al. [11] proposed the first data based approach to find influential users in a social network by considering the temporal aspect in the cascade of common actions performed by users, instead of using just static simulation of the friendship network. However, their work does not consider the time constraint in the information flow. In [10] they do use a time window based approach to determine true leaders in the network. However, the time window they consider is for direct influence only, i.e., once a user performs an action how many of his/her friends repeat that action in that time window. They have some additional assumptions like information propagation is non-cyclic and if one user performs an action more then once, they use only the time stamp of the first action. Our approach does not make such assumptions and identifies influential nodes without any constraints on the number of times a user performs an action or that the propagation graph needs to be a DAG. The time constraints we impose are on the path of information flow from the start of the action. Also, our proposed solu-

Table 1: Comparison of related work on different parameters

	Gomez-Rodriguez [24]	Cohen [6]	Du,N [8]	Tang [25]	Goyal [11, 10]	Kempe [13]	Lei [20]	IRS
Static Graph(S), Data or Cascade (C), Interaction Network (I)	C	S	C	S	C	S	S	I
Considers information channel or pathways?	Yes	No	Yes	No	Yes	No	No	Yes
Time window constrained	Yes	No	Yes	No	Yes	No	No	Yes
Approx sketching or sampling	Yes	Yes	Yes	Yes	No	No	Yes	Yes
One Pass algorithm	No	Yes	No	No	Yes	No	Yes	Yes

tion just needs a single pass over the propagation graph whereas Goyal's work do a single pass over the action log but multiple passes on the social network to find the child nodes. Our sketch based approximation further improves the time and space complexity.

There are a few more recent works on data driven approach by Gomez-Rodriguez et al. [24] and Du et al. [8]. These works try to derive the underlying hidden influence network and the influence diffusion probabilities along every edge from a given cascade of infection times for each node in the network. Du et al. [8] proposed a scalable algorithm called *ConTinEst*, which finds most influential nodes from the derived influence network. *ConTinEst* uses an adaption of a randomized neighborhood estimation algorithm [5] to find the most influential node in the network. But getting the cascade data of infection times for every network is not always possible. For example in an email or a messaging network, we may have access only to interactions between the users and not to the actual individual infection time. To the best of our knowledge our work is the first to try to predict and maximize influence in a network in which only the interaction data is available and no other action cascade or relationship between users is provided.

In Table 1 we give a brief comparison matrix of our IRS approach with some of the other works in Influence Maximization. We compare against the type of input each approach considers; i.e, a static graph (S), action cascade or infection time based event cascades (C) or interaction network based (I). We also compare if in the modeling of the information propagation in the approach considers information pathways or channels to do influence maximization and if the pathways have time window based constrains. For performance comparison, we see if they do use some sampling or sketching techniques to improve performance and if the algorithm is a one pass algorithm.

6. EXPERIMENTAL EVALUATION

In this section, we address the following questions:

Accuracy of Approximation. How accurate is the approximation algorithm for the Oracle problem? In other words, how well can we estimate the size of the IRS set based on the versionned HLL sketch?

Efficiency. How efficient is the approximate algorithm in terms of processing time per activity, and how does the window length ω impact the efficiency? How long does it take to evaluate an Oracle query based on the IRS summary?

Effectiveness. How effective is the identification of influential nodes using IRS to maximize the influence spread under the Time-Constrained Information Cascade Model? To this end, we compare our algorithm to a number of competitors:

- SKIM is the only algorithm which scale to large datasets in few minutes time. We ran SKIM using the same parameters Cohen et al. [6] use in their paper for all the experiments. SKIM is from the category of algorithms which considers a static graph and takes input in the form of a DIAMICS format graph. Hence we convert the interaction network data into the required static graph format by removing repeated interactions and the time stamp of every interaction.

- ConTinEst(CTE) [8] is the latest data driven algorithm which works on static networks where the edge weights corresponds to the associated transmission times. The edge weight is obtained from a transmission function which in turn is derived from an cascade of infection time of every node. As we assume that only the interaction between different nodes of a network is being observed and no other information such as the Infection time cascade is available, we transform the interactions into a static network with edge weights as required by ConTinEst. The first time a node u appears as the source of an interaction we assign the infection time u_i for the source node as the interaction time. Then each interaction (u, v, t) is transformed into an weighted edge (u, v) with the edge weight as the difference of the interaction time and the time when the source gets infected, i.e, $t - u_i$. We ran the same code as published by the authors with the default settings on the transformed data.

- The popular baselines *PageRank(PR)* and *High Degree(HD)*[13]. Here we select the k nodes with respectively the highest PageRank and out-degree. Notice that for PageRank we reversed the direction of the interaction edges, as PageRank measures incoming "importance" whereas we need outgoing "influence." By reversing the edges this aspect is captured. To make a fair comparison with our algorithm that takes into account the overlap of the influence of the selected top-influencers, we developed a version of HD that takes into account overlap. That is, we select a set of nodes that *together* have maximal outdegree. In our experiments we call this method the Smart High Degree approach (SHD). Notice that SHD is actually a special case of our IRS algorithm, where we set $\omega = 0$.

We also ran some performance experiments comparing the competitors to our IRS algorithm. In the interpretation of these results, however, we need to take into ac-

Table 2: Characteristics of interaction network along with the time span of the interactions as number of days.

Dataset	$\lvert\mathcal{V}\rvert[.10^3]$	$\lvert\mathcal{E}\rvert[.10^3]$	Days
Enron	87.3	1,148.1	8,767
Lkml	27.4	1,048.6	2,923
Facebook	46.9	877.0	1,592
Higgs	304.7	526.2	7
Slashdot	51.1	140.8	978
US-2016	4,468	44,638	16

count that the static methods require the graph to be preprocessed and takes as input the flattened non-temporal graph, which is in some cases significantly smaller as it does not take repetitions of activities into account.

6.1 Datasets and Setup

We ran our experiments on real-world datasets obtained from the SNAP repository [18] and the koblenx network collection [16]. We tested with *social* (Slashdot, Higgs, Facebook) and *email* (Enron, Lkml) networks. As the real world interaction networks available from previous works were not large enough to test the scalability of our algorithm, we created another dataset by tracking tweets related to the US Election 2016. We follow the same technique used to create the Higgs data set of the SNAP repository. Statistics of these data sets are reported in Table 2. These datasets are available online, sorted by time of interaction. We kept the datasets in this order, as our algorithm assumes that the interactions are ordered by time. This assumption is reasonable in real scenarios because the interactions will always arrive in increasing order of time and it is hence plausible that they are stored as such. The overall time span of the interactions varies from few days to many years in the data sets. Therefore, in our experiments we express the window length as a percentage of the total time span of the interaction network.

The performance results presented in this section are for the C++ implementation of our algorithm. All experiments were run on a simple desktop machine with an Intel Core i5-4590 CPU @3.33GHz CPU and 16 GB of RAM, running the Windows 10 operating system. For the larger dataset US-2016 the memory required was more than 16 GB. hence, we ran the experiments for the US-2016 dataset on a Linux system with 64 GB of RAM.

6.2 Accuracy of the Approximation

In order to test the accuracy of the approximate algorithm, we compared the algorithm with the exact version. We compute the average relative error in the estimation of the *IRS* size for all the nodes, in function of the number of buckets ($\beta = 2^k$). Running the exact algorithm is infeasible for the large datasets due to the memory requirements, and hence, we test only on the Slashdot and Higgs datasets to measure accuracy. We tested accuracy at different window lengths. The results are reported in Table 3. As expected from previous studies, the accuracy increases with β. There is a decrease in accuracy with increasing window length because as the window length increases, the number of nodes with larger *IRS* increases as well, resulting in a higher average error. β values beyond 512 yield only modest further improvement in the accuracy. Therefore, we used $\beta = 512$ as default for all of the next experiments.

Table 3: Average relative error in the estimation of the *IRS* size for all the nodes as a function of b for different window length.

Dataset	β	window %		
		1	10	20
Higgs	16	0.075	0.116	0.113
	32	0.044	0.081	0.053
	64	0.026	0.056	0.046
	128	0.008	0.015	0.017
	256	0.005	0.008	0.009
	512	0.002	0.006	0.007
Slashdot	16	0.048	0.055	0.105
	32	0.023	0.044	0.042
	64	0.013	0.022	0.33
	128	0.011	0.04	0.05
	256	0.01	0.026	0.025
	512	0.005	0.019	0.02

Table 4: Memory used in MB to process all the interactions at different window length ω

Datasets	$\omega = 1$	$\omega = 10$	$\omega = 20$
Slashdot	194.9	385.4	431.5
Higgs	1008.6	1138.3	1229.8
Enron	416.3	426	426.3
Facebook	247.4	470	496.2
Lkml	228.5	282.5	295.2
US-2016	50,449	56,829	59,104

6.3 Runtime and Memory usage of the Approximation Algorithm

We study the runtime of the approximation algorithm on all the datasets for different window lengths ω. The runtime increases with the increasing window length, as expected given that the number of nodes in the *IRS* increases, resulting in more elements in the vHLL to be merged. We study the processing time in function of the time window ω. Here we vary ω from 1% to 100%. The results are reported in Figure 3. It is interesting to see in Figure 3 that the processing time becomes almost constant as soon as the window length reaches 10%. This is because the *IRS* does not change much once the time window is large enough. This behavior indicates that at higher window lengths the analysis of the interaction network becomes similar to that of the underlying static network. As the algorithm is one pass it scales linearly with the input size. For the largest data set US-2016 with approx 45 million interactions the algorithm was able to parse all the interactions in just 8 min.

As shown in Table 4, we observe that the space consumption is essentially dependent on the number of nodes and not on the number of interactions on the network. For example, on Enron dataset the total space requirement is just 295 MB for $\omega = 20\%$, whereas for Higgs the memory requirement is 1229 MB, as the number of nodes for this data set is 4 times that of Enron. It is natural to see a slight increase in the space requirement with window length ω as the lists in the vHLL sketches become larger.

6.4 Influence Oracle Query Efficiency

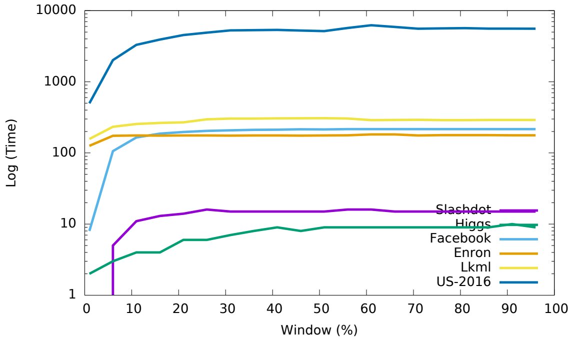


Figure 3: Log of the time to process all the interactions as a function of time window ω

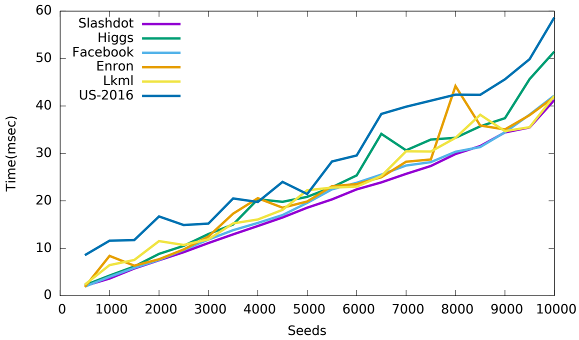


Figure 4: Influence spread prediction query time in milliseconds for window length, $\omega = 20\%$ as a function of the seed set size.

Now, we present the query time for the Influence Oracle using *IRS*. After the pre-processing step of computing the *IRS* for all nodes, querying the data structure is very efficient. We pick seed nodes randomly and query the data structure to calculate their combined influence spread. In Figure 4 we report the average query time for randomly selected seeds. We observe that, irrespective of the graph size the query time is mostly the same for all graphs. This is because the complexity of the versionned HyperLogLog union is independent of the set size. As expected, query time increases with the number of seed nodes. Even for numbers of seed nodes as large as 10,000, the query time is just few milliseconds.

6.5 Influence Maximization

Our next goal is to study how the *Influence Reachability Set* could be used to solve the problem of Influence Maximization. First we do an effectiveness analysis and then an efficiency comparison with the baseline approaches.

Effectiveness analysis:

We compare the influence spread by running the Time Constrained Information Cascade Model with infection probabilities of 50% and 100%. We compare our sketch based algorithm with the latest sketch based probabilistic approach SKIM [6] and ConTinEst(CTE) [8]. As Both SKIM and ConTinEst require a specific input format of the underlying static graph we ran a pre-processing phase to generate the required graph data from the interaction network. We ran both SKIM and ConTinEst using the code published by the respective authors. We also compare with other popular baselines *PageRank(PR)* and *High Degree(HD)*[13] by selecting top k nodes with highest page rank and highest out degree. We used 0.15 as the restart probability and a difference of 10^{-4} in the $L1$ norm between two successive iterations as the stopping criterion. We also introduced a variation of High Degree called *Smart High Degree(SHD)* in which instead of selecting top k nodes with highest degree we select nodes using a greedy approach to maximize the distinct neighbors.

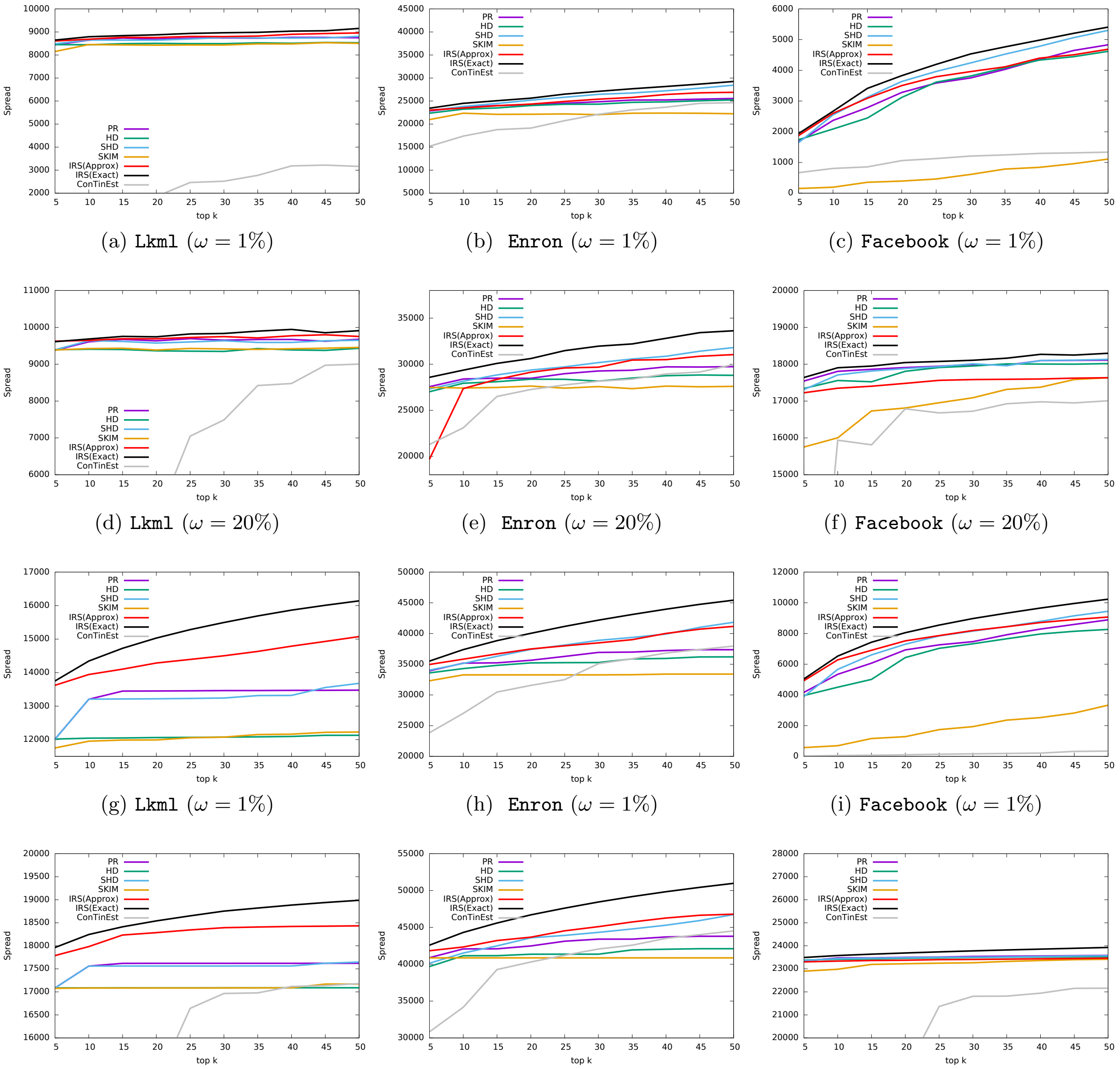
Table 5: Common seeds between different window length for top 10 seeds

Datasets	1% - 10%	1% - 20%	10% - 20%
`Slashdot`	0	0	7
`Higgs`	3	1	3
`Enron`	0	0	6
`Facebook`	4	4	9
`Lkml`	1	0	5
`US-2016`	6	6	10

The results of our comparison are reported in Figure 5. We observe that in all the datasets the influence spread by simulation through the seed nodes selected by our IRS exact algorithm is consistently better than that of other baselines. The IRS approx approach results in lesser spread but still it is best for `Lkml` dataset and is close to other baselines in other datasets. In other datasets like `Enron` or `Facebook` the nodes with highest degree are the same node for which the longer temporal paths exists hence the spread is similar. SKIM and ConTinEst both perform worst at smaller windows but with higher window lengths their performance increases; this is because for higher window lengths there is less pruning of the information channels resulting in a very small change in the Influence reachability set size. Hence, the behavior is the same as the analysis of the static graph and the time window does not have much effect on the *Influence Reachability Set*. The Smart High Degree approach out-performs High Degree in all of the cases. For smaller values of k the spread is very similar because of common seeds, for example 4 out of 5 seeds are common in `Slashdot` as nodes with highest page Rank is the also the node with highest degree and highest IRS set size at $\omega = 1\%$. But as k increases IRS performs much better.

Efficiency analysis:

Next, we compared the time required to find the top 50 seeds. The results are reported in Table 6. For IRS we report time taken by the more efficient IRS approx approach. The IRS approach takes more time for `Enron` and `Lkml` as compare to other baselines because the IRS approach depends on the number of interactions. While IRS is slower than Page Rank and Smart High Degree for smaller datasets it scales linearly with the size and takes 8 times less time for the `US-2016` dataset with millions of nodes and interactions. For SKIM the time required to find top k seeds is quite low. However, it requires preprocessed data in the DIMACS graph format [1] and the pre-processing step takes up to 10 hours for the `US-2016` dataset. ConTinEst does not scale so well for large graphs and is the slowest in all dataseta. For the `US-2016` dataset the memory requirements were so high that it could not even finish the processing. IRS provides a promising tradeoff between efficiency and effectiveness, especially for smaller window lengths when the tempo-



(a) Lkml ($\omega = 1\%$) (b) Enron ($\omega = 1\%$) (c) Facebook ($\omega = 1\%$)

(d) Lkml ($\omega = 20\%$) (e) Enron ($\omega = 20\%$) (f) Facebook ($\omega = 20\%$)

(g) Lkml ($\omega = 1\%$) (h) Enron ($\omega = 1\%$) (i) Facebook ($\omega = 1\%$)

(j) Lkml ($\omega = 20\%$) (k) Enron ($\omega = 20\%$) (l) Facebook ($\omega = 20\%$)

Figure 5: Comparing the spread of the influence of top k seeds using Simulation Algorithm for different seed size at different window length ω at Infection probability 50%(a-f) and 100%(g-l) respectively.

Table 6: Time in seconds to find top 50 seeds by IRS(approx) and all other baseline approach.

Datasets	IRS	SKIM	PR	HD	SHD	CTE
Slashdot	1.1	1.2	21.9	0.9	2.1	694
Higgs	2.2	4.3	29.8	0.7	1.5	3,802
Enron	93.7	2.2	49.4	0.4	8.1	1,349
Facebook	10.3	1.1	35.6	0.5	2.9	790
Lkml	117.9	1.7	29.8	0.5	22.9	733
US-2016	498	23.6	4,261	47.4	3,338.4	-

ral nature of the graph has a higher role in determining the influential nodes.

Effect of window on top k seeds:

To see the effect of the time window on the most influential nodes we study the common seeds between different window lengths. We observed that the top k seeds change drastically as we change the window length, especially when the window length is small. But for window lengths greater than 10% the top k seeds do not change much. For US-2016 the top 10 seeds are exactly the same for the 10% and 20% window. In Table 5 we have reported the common seeds among different top 10 seeds at different window lengths. There are no common seeds between the top 10 seeds found for window lengths of 1% and 10% for Slashdot and Enron and only $3-4$ common seeds for Higgs, Facebook and Lkml. This shows that for different window lengths there are different nodes which become most influential and hence it is necessary to consider window length while doing Influence maximization.

7. CONCLUSION

We studied the problem of information propagation in an interaction network—a graph with a sequence of time stamped interactions. We presented a new time constrained *influence channel* based approach for Influence Maximization and Information Spread Prediction. We presented an exact algorithm, which is memory inefficient, but it set the stage for our main technique, an approximate algorithm based on a modified version of HyperLogLog sketches, which requires logarithmic memory per network node, and has fast update time. One interesting property of our sketch is that the query time of the Influence Oracle is almost independent of the network size. We showed that the time taken to do influence maximization by a greedy approach on our sketch is very time efficient. We also showed the effect of the time window on the influence spread. We conclude that smaller window lengths have very high impact on the Information propagation and hence it is important to consider the spread window to do Influence maximization.

8. REFERENCES

[1] 9th DIMACS Implementation Challenge - Shortest Paths. http://www.dis.uniroma1.it/challenge9/format.shtml#graph, [Online; accessed 12-Sep-2016]

[2] Chen, W., Lu, W., Zhang, N.: Time-critical influence maximization in social networks with time-delayed diffusion process. arXiv:1204.3074 (2012)

[3] Chen, W., Wang, C., Wang, Y.: Scalable influence maximization for prevalent viral marketing in large-scale social networks. In: Proceedings of the 16th ACM SIGKDD. pp. 1029–1038. ACM (2010)

[4] Chen, W., Wang, Y., Yang, S.: Efficient influence maximization in social networks. In: Proceedings of the 15th ACM SIGKDD. pp. 199–208. ACM (2009)

[5] Cohen, E.: Size-estimation framework with applications to transitive closure and reachability. Journal of Computer and System Sciences 55(3), 441–453 (1997)

[6] Cohen, E., Delling, D., Pajor, T., Werneck, R.F.: Sketch-based influence maximization and computation: Scaling up with guarantees. In: Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management. pp. 629–638. ACM (2014)

[7] Domingos, P., Richardson, M.: Mining the network value of customers. In: Proceedings of the seventh ACM SIGKDD. pp. 57–66. ACM (2001)

[8] Du, N., Song, L., Gomez-Rodriguez, M., Zha, H.: Scalable influence estimation in continuous-time diffusion networks. In: Advances in neural information processing systems. pp. 3147–3155 (2013)

[9] Flajolet, P., Fusy, É., Gandouet, O., Meunier, F.: Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. DMTCS Proceedings (2008)

[10] Goyal, A., Bonchi, F., Lakshmanan, L.V.: Discovering leaders from community actions. In: Proceedings of the 17th ACM conference on Information and knowledge management. pp. 499–508. ACM (2008)

[11] Goyal, A., Bonchi, F., Lakshmanan, L.V.: A data-based approach to social influence maximization. Proceedings of the VLDB Endowment 5(1), 73–84 (2012)

[12] Kempe, D., Kleinberg, J., Kumar, A.: Connectivity and inference problems for temporal networks. In: Proceedings of the thirty-second annual ACM symposium on Theory of computing. pp. 504–513. ACM (2000)

[13] Kempe, D., Kleinberg, J., Tardos, É.: Maximizing the spread of influence through a social network. In: Proceedings of the ninth ACM SIGKDD. pp. 137–146. ACM (2003)

[14] Kleinberg, J.: The flow of on-line information in global networks. In: Proceedings of the 2010 ACM SIGMOD. pp. 1–2. ACM (2010)

[15] Kumar, R., Calders, T., Gionis, A., Tatti, N.: Maintaining sliding-window neighborhood profiles in interaction networks. In: Machine Learning and Knowledge Discovery in Databases, pp. 719–735. Springer (2015)

[16] Kunegis, J.: Konect: the koblenz network collection. In: Proceedings of the 22nd international conference on World Wide Web companion. pp. 1343–1350. International World Wide Web Conferences Steering Committee (2013)

[17] Leskovec, J., Krause, A., Guestrin, C., Faloutsos, C., VanBriesen, J., Glance, N.: Cost-effective outbreak detection in networks. In: Proceedings of the 13th ACM SIGKDD. pp. 420–429. ACM (2007)

[18] Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data (Jun 2014)

[19] Leskovec, J., McGlohon, M., Faloutsos, C., Glance, N.S., Hurst, M.: Patterns of cascading behavior in large blog graphs. In: SDM. vol. 7, pp. 551–556. SIAM (2007)

[20] Liu, B., Cong, G., Xu, D., Zeng, Y.: Time constrained influence maximization in social networks. In: Data Mining (ICDM). pp. 439–448. IEEE (2012)

[21] Mohammadi, A., Saraee, M., Mirzaei, A.: Time-sensitive influence maximization in social networks. Journal of Information Science 41(6), 765–778 (2015)

[22] Prakash, B.A., Chakrabarti, D., Valler, N.C., Faloutsos, M., Faloutsos, C.: Threshold conditions for arbitrary cascade models on arbitrary networks. Knowledge and information systems 33(3), 549–575 (2012)

[23] Richardson, M., Domingos, P.: Mining knowledge-sharing sites for viral marketing. In: Proceedings of the eighth ACM SIGKDD. pp. 61–70. ACM (2002)

[24] Rodriguez, M.G., Schölkopf, B.: Influence maximization in continuous time diffusion networks. arXiv:1205.1682 (2012)

[25] Tang, Y., Shi, Y., Xiao, X.: Influence maximization in near-linear time: A martingale approach. In: Proceedings of the 2015 ACM SIGMOD. pp. 1539–1554. ACM (2015)

[26] Wu, H., Cheng, J., Huang, S., Ke, Y., Lu, Y., Xu, Y.: Path problems in temporal graphs. Proceedings of the VLDB Endowment 7(9), 721–732 (2014)

Flexible Caching in Trie Joins

Oren Kalinsky Yoav Etsion Benny Kimelfeld
Technion – Israel Institute Of Technology
{okalinsk@campus, yetsion@tce, bennyk@cs}.technion.ac.il

ABSTRACT

While traditional algorithms for multiway join are based on reordering binary joins, more recent approaches have instantiated a new breed of "worst-case-optimal" in-memory algorithms wherein all relations are scanned simultaneously. Veldhuizen's Leapfrog Trie Join (LFTJ) is an example. An important advantage of LFTJ is its small memory footprint, due to the fact that intermediate results are full tuples that can be dumped immediately. However, since the algorithm does not store intermediate results, recurring joins must be reconstructed from the source relations, resulting in excessive memory traffic. In this paper, we address this problem by incorporating caches into LFTJ. We do so by adopting recent developments in join optimization, tying variable ordering to a tree decomposition of the query. While the traditional usage of tree decomposition computes the entire result for each bag, our proposed approach incorporates caching directly into LFTJ and can dynamically adjust the size of the cache. Consequently, our solution balances between memory usage and repeated computation. Our experimental study over the SNAP dataset compares between various (traditional and novel) caching policies, and shows significant speedups over state-of-the-art algorithms on both join evaluation and join counting.

CCS Concepts

•**Information systems** → **Join algorithms;** Query optimization; *Main memory engines;*

Keywords

Databases, trie joins, tree decomposition, caching

1. INTRODUCTION

Traditional optimization of multiway joins has been based on decomposing the query into smaller join queries, and combining intermediate relations. This approach has roots in Selinger's pairwise-join enumeration [26], and it includes the application of the algorithm of Yannakakis [30] over a tree decomposition of the query [13, 14]. Recent approaches have developed a new breed of in-memory algorithms wherein all relations are scanned simultaneously [1, 10, 15, 16, 21, 28, 29], featuring the complexity guarantee of *worst-case optimality*. This yardstick of efficiency has been introduced by Ngo et al. [21], and it states that for every join query, no algorithm can be asymptotically faster on the space of all databases; in that work they presented the first worst-case optimal algorithm, later termed *NPRR* [21]. Effectively, the running time is bounded by the *AGM bound* [5] that determines the maximal number of tuples in the multiway join of relations with given sizes.

Leapfrog Trie Join (LFTJ) [29] is another worst-case-optimal algorithm, introduced by LogicBlox and implemented in the company's product [3]. It operates in a manner of *variable elimination* where there is a linear order over the variables, and query results are generated one by one by incrementally assigning values to each variable in order. Trie-structured indices over the relations allow to efficiently determine whether the next variable in consideration can be assigned a value that is consistent with the assignments to the previous variables. (We give a detailed description of LFTJ in Section 2). Beyond being worst-case optimal, LFTJ has two important features. First, it avoids the potential generation of intermediate results that may be substantially larger than the final output size (which is a key property in guaranteeing worst-case optimality). Second, LFTJ is very well suited for in-memory join evaluation, since besides the trie indices it has a close to zero memory consumption. Of course, memory is required for buffering the tuples in the final result, but these are never read and can be safely dumped to higher storage upon need. Moreover, these tuples are not even needed in the case of common aggregate queries (e.g., count the number of tuples in the result).

Yet, intermediate results have the advantage that their tuples can be reused, and this is especially substantial in the presence of a significant skew. In our experiments, we have found that LFTJ often loses its advantage to the built-in caching of intermediate results of the traditional approaches, and in particular, LFTJ is often required to apply many repetitions of computations. The repeated traversals back and fourth on the trie index generate excessive memory traffic, which has detrimental impact on the performance of database systems [2]. For example, our analysis of the memory load induced by LFTJ found that running a single count 5-cycle query on the SNAP ca-GrQc dataset generates over $45 \cdot 10^9$ memory accesses, whereas running the same query using tree decomposition and Yannakakis's join generates less than $16 \cdot 10^9$ accesses. (The implementation of both algorithms is discussed in Section 5.)

Our goal in this work is to accelerate LFTJ by incorporating caching in a way that (a) allows for computation reuse, and (b) does not compromise its key advantages. In particular, our goal is to incorporate caching in LFTJ so that it can utilize whatever memory it has at its disposal towards memoization. However, it is not clear how LFTJ can cache intermediate results (without com-

Series ISSN: 2367-2005

10.5441/002/edbt.2017.26

puting and storing full results of subqueries as done in other algorithms [1, 28]). Intuitively, the challenge lies in the fact that every iteration involves a different partial assignment, and variables are interdependent through the query structure. Our solution is inspired by recent developments in the theory of join optimization, relating to worst-case optimality and tree decomposition [15, 16, 28]. But unlike existing work, we do not apply the join algorithm on each bag independently (which would result in high memory consumption due to intermediate results), but rather execute LFTJ as originally designed.

Specifically, to enable effective caching our approach applies the following steps. We first build a Tree Decomposition (TD) for the query, in a manner that we discuss later on. Intuitively, a TD transforms the query into a tree structure by grouping together several relations, where each group is called a *bag*. We then execute LFTJ as usual, but throughout the execution we use caches (deploying a caching/eviction policy) for partial assignments. More formally, each bag of the TD is assigned a cache, and the application of the cache happens when the iteration over the variables enters a new bag. The correctness of the cache usage (i.e., the fact that the intermediate assignments are consistent with the current assignment in construction) is crucially based on two properties.

1. The variable ordering is required to be *compatibile* with the TD. Intuitively, compatibility means that the variable order is consistent with the preorder of the TD. (The formal definition is in Section 2.)
2. Each cache applies to partial assignments only for the variables it contains (for evaluation) or the subtree underneath (for counting).

For TD computation, there is a plethora of algorithms with different *quality* guarantees. The classical graph-theoretic measure refers to the maximal size of a bag, and a generalization to hypergraphs is based on the notion of a *hypertree width*. The optimal values of those (i.e., realizing the *tree width* and the *hypertree width*, respectively) are both NP-hard problems [4, 13], and efficient algorithms exist for special cases and different approximation guarantees [8]. Other notions include decompositions that approximate the minimal *fractional hypertree width* [15, 19]. In our case, a TD defines a caching scheme, and various factors determine the effectiveness of this scheme. Caches are more reusable in the presence of skewed data, and hence, data statistics can be used to estimate the goodness of a TD. Importantly, our caches correspond to the *adhesions* (parent-child intersections); in order to better capture opportunities of a high skew (and a high hit rate), we give precedence to keys from a domain of a smaller dimension, and hence, we favor smaller adhesions. Due to these arguments, we chose not to use any specific algorithm that generates a single tree decomposition, but rather to explore *a large space of such decompositions*. We devise a heuristic algorithm for enumerating TDs, tailored primarily towards small adhesions. Once such a collection of TDs are generated, we deploy a cost function that takes various factors into account, including the skew-based cost model of Chu et al. [10].

We experiment on three types of queries: paths, cycles and random. In par with recent studies on join algorithms, we base our experiments on datasets from the SNAP [18] and IMDB workloads. We explore several attributes of our cached LFTJ, such as the cache size and the eviction policy. We also experiment with the count version of the queries. Our experiments compare among LFTJ, with and without caching, and Yannakakis's algorithm over the TD (as in DunceCap [24,28]), as well as other various systems and engines (LogicBlox [3], PostgreSQL [27] and EmptyHeaded [1]). The results show consistent improvement compared to LFTJ (in orders of magnitude on large queries), as well as general improvement compared to the examined algorithms and systems. The only alternative that outperforms our implementation on a large portion of the count queries is EmptyHeaded, as it implements a parallel implementation using the *Single Instruction Multiple Data* (SIMD) parallelization model (while our implementation applies standard sequential computation). We defer hardware utilization of this sort to future research.

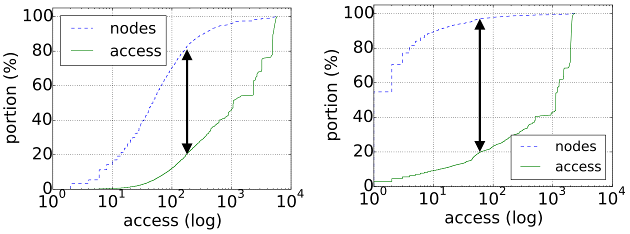


Figure 1: Mass-count disparity plots for value accesses on the evaluation of a 5-path (left) and a 5-cycle (right) over the SNAP ca-GrQc dataset; the double-headed arrows indicate that 80% of the accesses are applied to just 20% (left) and 5% (right) of the nodes

While retaining the inherent features of LFTJ, our caching dramatically reduces the memory accesses. For illustration, running a 5-cycle count query generates only $1.4 \cdot 10^9$ memory accesses, which is over 30× fewer accesses than vanilla LFTJ (and over 10× fewer accesses than TD with Yannakakis's algorithm). Figure 1 provides some intuition on why we are able to establish such a dramatic improvement with a modest memory usage. The figure depicts *mass-count disparity plots* [11] for value accesses on the evaluation of a 5-path (left) and a 5-cycle (right) over the SNAP ca-GrQc dataset, which has a graph structure. The x-axis corresponds to the number of accesses. A tick at number n refers to the nodes that are accessed *at most* n times by our algorithm (node popularity); the dashed curve shows the fraction of such nodes among all nodes, and the solid curve shows the fraction of accesses to such nodes among all accesses. On the left plot we can see, for example, that 80% of the accesses are directed to around 20% of the most popular nodes (as indicated by the double-headed arrow), and on the right one we can see that 80% of the accesses are applied to 5% of the most popular nodes!

To summarize, our contributions are as follows. First, we extend LFTJ with caching, without compromising the key benefits. Our caching is executed alongside LFTJ, and its size can be determined dynamically according to memory availability. This is achieved by combining LFTJ with a TD, a suitable variable ordering, and a suitable set of target variables for each cache. Second, we devise a heuristic approach to enumerating tree decompositions of a CQ; this approach favors small adhesions, and is based on enumerating graph separating sets by increasing size. Third, we present a thorough experimental study that evaluates the effect of caching on LFTJ, on both evaluation and counting, and compares the results to state-of-the-art join algorithms.

2. BACKGROUND

In this section we give preliminary definitions and notation that we use throughout the paper.

2.1 Conjunctive Queries

We study the problem of *evaluating* a Conjunctive Query (CQ), and the problem of *counting* the number of tuples in the result of a CQ. As in recent work on worst-case optimal joins [21, 22, 29], we focus here on *full CQs*, which are CQs without projection. Formally, a full CQ is a sequence $\varphi_1, \dots, \varphi_m$ where each φ_i is a *subgoal* of the form $R(\tau_1, \dots, \tau_k)$ with R being a k-ary relation name and each τ_j being either a constant or a variable. In the remainder of this paper, we say simply "CQ" instead of "full CQ." We denote by $\mathsf{vars}(\varphi_j)$ the set of variables that occur in φ_j, and we denote by $\mathsf{vars}(q)$ the union of the sets $\mathsf{vars}(\varphi_j)$ over all atoms φ_j in q (i.e., the set of all variables appearing in q).

Let q be a CQ. A *partial assignment* for q is function μ that maps every variable in $\mathsf{vars}(q)$ to either a constant value or *null* (denoted $\bot$). If μ is a partial assignment for q, then we denote by $q_{[\mu]}$ the CQ that is obtained from q by replacing every variable x with $\mu(x)$, if $\mu(x) \neq \bot$, and leaving x intact if $\mu(x) = \bot$. If X is a subset of $\mathsf{vars}(q)$, then we denote by $\mu_{|X}$ the restriction of μ to X; that is, $\mu_{|X}$ is defined only over X, and $\mu_{|X}(x) = \mu(x)$ for all $x \in X$.

For a CQ q, a partial assignment that maps every variable to a (nonnull) constant is called a *complete* assignment. Let D be a database over the same relation names as q. *Evaluating* q over D is the task of producing the set $q(D)$, which consists of all complete assignments μ such that all the ground subgoals of $q_{[\mu]}$ are facts (tuples) of D; such an assignment is also called an *answer* (for q over D). *Counting* q over D is the task of computing the number of answers, that is, $|q(D)|$.

The *Gaifman graph* of a CQ q is the undirected graph that has $\mathsf{vars}(q)$ as its node set and an edge between every two variables that co-occur in a subgoal of q.

EXAMPLE 2.1. Our running example uses the following CQ q over a single binary relation R.

$$R(x_1, x_2), R(x_2, x_3), R(x_2, x_4), R(x_3, x_4), R(x_3, x_5), R(x_4, x_6)$$

Observe that q does not have constant terms. This CQ is illustrated in the graph of Figure 2(a); in this case the graph is also the Gaifman graph of q (since q is binary). The graph is also the Gaifman graph of the following CQ:

$$R(x_1, x_2), S(x_2, x_3, x_4), R(x_3, x_4), R(x_3, x_5), R(x_4, x_6)$$

Let μ be the partial assignment that maps x_1 and x_2 to the constants 1 and 2, respectively, and the other variables to $\bot$. Then $q_{[\mu]}$ is

$$R(1, 2), R(2, x_3), R(2, x_4), R(x_3, x_4), R(x_3, x_5), R(x_4, x_6)\,.$$

Our example database D, depicted in Figure 2(b), consists of a single relation. It can verified that $q(D)$ contains the following assignments μ_1 and μ_2:

- μ_1: $x_1 \mapsto 1$, $x_2 \mapsto 2$, $x_3 \mapsto 1$, $x_4 \mapsto 2$, $x_5 \mapsto 3$, $x_6 \mapsto 1$
- μ_2: $x_1 \mapsto 1$, $x_2 \mapsto 2$, $x_3 \mapsto 2$, $x_4 \mapsto 1$, $x_5 \mapsto 1$, $x_6 \mapsto 3$

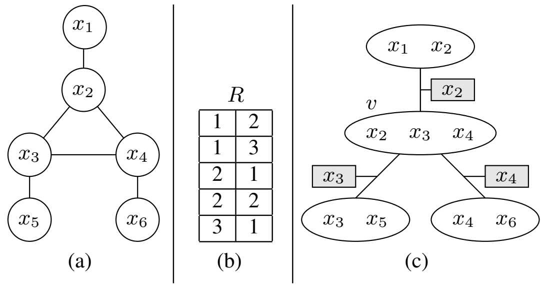
If we remove from μ_1 and μ_2 the assignments for x_1 and x_2, then we get answers in $q_{[\mu]}(D)$ for the above defined μ. □

2.2 Ordered Tree Decompositions

Let $q = \varphi_1, \dots, \varphi_m$ be a CQ. A *Tree Decomposition* (*TD*) of q is a pair $\langle t, \chi \rangle$ where t is a tree and χ is a function that maps every node of t to a subset $\chi(v)$ of $\mathsf{vars}(q)$, called a *bag*, such that both of the following hold.

- For each φ_j there is a node v of t with $\mathsf{vars}(\varphi_j) \subseteq \chi(v)$.
- For each x in $\mathsf{vars}(q)$, the nodes v with $x \in \chi(v)$ induce a connected subtree of t.

An *ordered TD* of a CQ q is pair $\langle t, \chi \rangle$ defined similarly to a TD, except that t is a rooted and ordered tree. We denote the root of t by $\mathrm{root}(t)$. Let v be a node of t. We denote by $t_{|v}$ the subtree of t that is rooted at v and contains all of the descendants of v. If v is a non-root node, then the *parent adhesion* of v (or simply the adhesion *of* v) is the set $\chi(p) \cap \chi(v)$ where p is the parent of v, and is denoted by $\mathrm{adhesion}(v)$. Every set $\mathrm{adhesion}(u)$, where u is a non-root node of t, is called an *adhesion* of $\langle t, \chi \rangle$.



1	2
1	3
2	1
2	2
3	1

Figure 2: (a) Example of a CQ q; (b) A database D with a single relation; (c) An ordered tree decomposition of q

EXAMPLE 2.2. We continue with our running example. Figure 2(c) depicts an ordered tree decomposition $\langle t, \chi \rangle$ of the query q of Figure 2(a). The tree t has four nodes, and the order is top down, left to right. The root is the top node with the bag $\{x_1, x_2\}$. To verify that it is indeed a tree decomposition of q, the reader needs to check that every edge in Figure 2(a) is contained in some bag of $\langle t, \chi \rangle$. The adhesions of $\langle t, \chi \rangle$ are shown in the gray boxes. Let v be the node of t with $\chi(v) = \{x_2, x_3, x_4\}$. The parent adhesion of v, which we denote by $\mathrm{adhesion}(v)$, is the singleton $\{x_2\}$. □

Let q be a CQ, and let $\langle t, \chi \rangle$ be an ordered TD of q. The *preorder* of t is the order $\prec$ over the nodes of t such that for every node v with a child c preceding another child c', and nodes u and u' in $t_{|c}$ and $t_{|c'}$, respectively, we have $v \prec u \prec u'$. We denote the preorder of t by $\prec_{\mathsf{pre}}$. For a variable x in $\mathsf{vars}(q)$, the *owner bag* of x, denoted $\mathrm{owner}(x)$, is the minimal node v of t, under $\prec_{\mathsf{pre}}$, such that $x \in \chi(v)$. For a node v of t, we denote by $\mathrm{owned}(v)$ the set of variables x that have v as the owner. We say that $\langle t, \chi \rangle$ is *compatible* with an ordering $\langle x_1, \dots, x_n \rangle$ if $i < j$ whenever $\mathrm{owner}(x_i) \prec_{\mathsf{pre}} \mathrm{owner}(x_j)$. We may also say that the ordering $\langle x_1, \dots, x_n \rangle$ is compatible with $\langle t, \chi \rangle$ if the latter is compatible with the former.

EXAMPLE 2.3. Consider the given ordering $\langle x_1, \dots, x_6 \rangle$ of the variables in our running example (Figure 2), and the TD $\langle t, \chi \rangle$ of Figure 2(c). The preorder of t is given by $\{x_1, x_2\}$, $\{x_2, x_3, x_4\}$, $\{x_3, x_5\}$, $\{x_4, x_6\}$. We have $\mathrm{owner}(x_3) = \mathrm{owner}(x_4) = v$, and $\mathrm{owned}(v) = \{x_3, x_4\}$. Note that $\mathrm{owner}(x_2) \neq v$ since x_2 occurs already in the root of t (and therefore $\mathrm{owner}(x_2) = \mathrm{root}(t)$). □

2.3 Trie Join

We now describe the Leapfrog Trie Join (LFTJ) algorithm [29]. Our description is abstract enough to apply to the *tributary join* of Chu et al. [10]. Let $q = \varphi_1, \dots, \varphi_m$ be a CQ. The execution of LFTJ is based on a predefined ordering $\langle x_1, \dots, x_n \rangle$ of $\mathsf{vars}(q)$. The correctness and theoretical efficiency of LFTJ are guaranteed on every order of choice, but in practice the order may have a substantial impact on the execution cost [10]. Moreover, in our instantiation of LFTJ we will use orderings with specific properties.

For every subgoal φ_k, LFTJ maintains a trie structure on the corresponding relation r. Each level i of the trie corresponds to a variable x_j in $\mathsf{vars}(\varphi_k)$, and holds values that can be matched against x_j. Whenever x_j is in a level above $x_{j'}$ it holds that $j < j'$. Moreover, every path from root to leaf corresponds to a unique

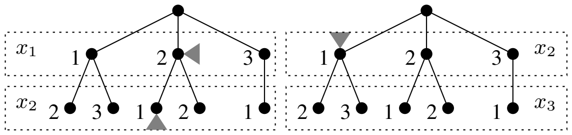


Figure 3: The trie structures for subgoals $R(x_1, x_2)$ and $R(x_2, x_3)$, respectively, in the running example

tuple of r and vice versa. Sibling values in the trie are stored in a sorted manner.

EXAMPLE 2.4. Figure 3 depicts two of the tries used for evaluating the CQ q of Example (2.1), of our running example. The left trie is for $R(x_1, x_2)$ and $R(x_2, x_3)$. (The reader should ignore the gray triangles for now; we discuss them in the next example.) In this case, the tries are identical (as they index the same relation), but they are used differently during query evaluation. Each level of the trie corresponds to a variable and a corresponding attribute. The path root→1→2 corresponds to the tuple $R(1, 2)$, and root→2→1 corresponds to $R(2, 1)$. □

LFTJ applies a sequence of unary joins, called *leapfrog joins*, as follows. Each trie holds an iterator, initialized by pointing to the root. A mapping μ, which is initialized with nulls, is maintained throughout the execution. First, all the subgoals that contain x_1 advance their iterators in the first level until a matching value a is found (i.e., all iterators point to a), and $\mu(x_1)$ is set to a. The matching value is found efficiently in a technique referred to as *leapfrogging* [29]. The algorithm then proceeds recursively[1] with the CQ $q_{[\mu]}$ by proceeding to the next matching value, and so on, until all variables are assigned values (and then μ is printed) or no matching values are found; then backtracking takes place by advancing the previous iterator. A balanced-tree storage of the sibling collections in the tries guarantees that alignment of the iterators on matching attributes is done efficiently (in an amortized sense), which in turn guarantees that LFTJ is *worst-case optimal* [21].

EXAMPLE 2.5. Continuing Example 2.4, the gray triangles in Figure 3 show a possible positioning of the pointers on the tries during the execution. Here, the pointer for x_1 is set on 2, the pointers of x_2 in both tries is set on 1 (which is a matching value found), and next a matching value for x_3 will be sought in under the pointed node in the right trie (and the other tries). □

We refer the reader to the original publication [29] for more details on LFTJ. In this paper, it suffices to regard LFTJ abstractly as depicted in Figure 4. We call the algorithm of Figure 4 *trie join* and denote it by TrieJoin. This algorithm updates the global partial assignment μ using the subroutine RJoin (Recursive Join).

3. CACHING IN TRIE JOIN

In this section we devise an algorithm that incorporates caching within TrieJoin (Figure 4). We first discuss the intuition.

3.1 Intuition

The general idea is as follows. Let q be the evaluated CQ, and let $\langle x_1, \dots, x_n\rangle$ be vars(q) in the order of iteration. Consider a point in the iteration where we complete the assignment for $x_1, \dots, x_j$ ($j < n$), and suppose that we have already encountered the assignment for $x_i, \dots, x_j$ in the past for some i such that $1 < i < j$. We would like to be able to reuse the past assignments, at least for a few of the next variables, say $x_{j+1}, \dots, x_k$, instead of searching again for matches. Integrating simple memoization in the algorithm will not suffice. The problem is that the assignments for $x_{j+1}, \dots, x_k$ may depend not just on those for $x_i, \dots, x_j$, but rather on the assignments for variables in $x_1, \dots, x_{i-1}$, and so reusing past assignments may lead to incorrect results (false assignments).

Algorithm TrieJoin($q, \langle x_1, \dots, x_n\rangle, \mathcal{T}$)

```
for $d = 1, \dots, n$ do
    $\mu(x_d) := \perp$
RJoin(1)
```

Subroutine RJoin(d)

```
if $d = n + 1$ then
    print $\mu$
    return
for all matching values $a$ for $x_d$ in $\mathcal{T}$ do
    position $\mathcal{T}$ on $x_d \mapsto a$
    $\mu(x_d) := a$
    RJoin($d + 1$)
reset $x_d$ pointers in $\mathcal{T}$
```

Figure 4: Trie join

The above problem is avoided as follows. First, we deploy an ordered TD $\langle t, \chi\rangle$, and use an ordering $\langle x_1, \dots, x_n\rangle$ that is compatible with $\langle t, \chi\rangle$ (as defined in Section 2.2). Second, cache keys are assignments to sequences $x_i, \dots, x_j$ of variables *only* if the set $\{x_i, \dots, x_j\}$ is an adhesion of some node v of t. Finally, we cache assignments *only* for the variables $x_{j+1}, \dots, x_k$ that are owned by v. Due to the nature of the TD, we can rest assured that the assignments to $x_{j+1}, \dots, x_k$ are independent of the assignments to $x_1, \dots, x_{i-1}$ (once we know the assignments for $x_i, \dots, x_j$).

EXAMPLE 3.1. Consider again our running example around Figure 2. At some point in the execution of TrieJoin we construct the assignment μ with $\mu(x_1) = 1$ and $\mu(x_2) = 2$, and then continue to the rest of the variables in order. The next assignments we construct are $x_3 \mapsto 1$ and $x_4 \mapsto 2$. Once we are done with the complete assignments for the extended μ, we construct the assignments $x_3 \mapsto 2$ and $x_4 \mapsto 1$, and later on $x_3 \mapsto 2$ and $x_4 \mapsto 2$. Later in the execution, we encounter the assignment μ' with $\mu(x_1) = 2$ and $\mu(x_2) = 2$. Since adhesion$(x) = \{x_2\}$, we check to see whether there is a cache for $x_2 \mapsto 1$, and if so, then it tells us exactly where to position the pointers for x_3 and x_4 (which are the variables owned by v) in each of the possibilities (which are $(1, 2)$, $(2, 1)$, $(2, 2)$ and nothing else). We may similarly have a cache for $\{x_3\}$ (lower left adhesion) and for $\{x_4\}$ (lower right adhesion). □

Caching could be obtained by computing the complete join for every bag (using TrieJoin), and then joining the intermediate results using an algorithm for acyclic joins such as Yannakakis [30], as done in DunceCap [24, 28]. However, we wish to control the memory consumption and avoid storing the complete joins of subqueries. Our algorithm executes TrieJoin ordinarily, yet caches results during the execution based on a deployed caching policy.

COMMENT 3.2. Compatibility of the variable ordering with the TD has implications on the trie structures, which need to be consis-

[1] The actual algorithm of [29] is not recursive, but rather applies a single procedure call. Recursion simplifies our presentation.

Algorithm CacheTrieJoin$(q, \langle x_1, \dots, x_n \rangle, \langle t, \chi \rangle, \mathcal{T})$

```
1: for $d = 1, \dots, n$ do
2:     $\mu(x_d) := \bot$
3: for all nodes $v$ of $t$ do
4:     $cache_v := \emptyset$
5: CacheRJoin$(1)$
```

Subroutine CacheRJoin(d)

```
1: if $d = n + 1$ then
2:     print $\mu$
3:     return
4: $v := \mathrm{owner}(x_d)$
5: $\{x_l, x_{l+1}, \dots, x_k\} := \mathrm{owned}(v)$
6: $\alpha := \mathrm{adhesion}(v)$
7: if $v \neq \mathrm{root}(t)$ and $d = l$ then
8:     if $\mu_{|\alpha}$ is a cache hit in $cache_v$ then
9:         for all cached entries $\mu'$ in $cache_v(\mu_{|\alpha})$ do
10:            for $i = l, \dots, k$ do
11:                $\mu(x_i) := \mu'(x_i)$
12:            AdjustTries$(\mathcal{T}, \mu')$
13:            CacheRJoin$(k + 1)$
14:        reset $x_l, \dots, x_k$ pointers in $\mathcal{T}$
15:        return
16: for all matching values $a$ for $x_d$ in $\mathcal{T}$ do
17:     position $\mathcal{T}$ on $x_d \mapsto a$
18:     $\mu(x_d) := a$
19:     CacheRJoin$(d + 1)$
20:     if $v \neq \mathrm{root}(t)$ and $d = k$ then
21:         ApplyCachePolicy$(cache_v, \mu_{|\alpha}, \mu_{|\mathrm{owned}(v)})$
22: reset $x_d$ pointers in $\mathcal{T}$
```

Figure 5: TrieJoin **with caching**

tent with the variable ordering [29]. Therefore, similarly to EmptyHeaded [15], the design of our tries depends on the TD. As building the trie may take considerable time, our approach matches the scenario where the join is known in advance, but not the data (which is common in Web applications where queries arise due to user interaction with the UI). Another matching scenario is where the relations are narrow (e.g., graphs), and then we can compute in advance multiple trie structures (which is the design choice of EmptyHeaded [15]) and load the proper ones upon need. □

3.2 Algorithm

We now turn to a more formal description of our algorithm, which we call CacheTrieJoin, and is depicted in Figure 5. The algorithm extends upon the algorithm of Figure 4 in the sense that when no caching takes place, the two algorithms coincide. The algorithm takes as input a CQ q, a variable ordering $\langle x_1, \dots, x_n \rangle$, an ordered TD $\langle t, \chi \rangle$ that is compatible with $\langle x_1, \dots, x_n \rangle$, and a trie structure $\mathcal{T}$ for a database D. The algorithm prints all tuples in $q(D)$. The algorithm uses a cache, denoted $cache_v$, for every node v of t, for caching computed assignments for the variables owned by v. The algorithm CacheTrieJoin simply initializes a global partial assignment μ and each $cache_v$, and calls the subroutine CacheRJoin (the caching version of RJoin of Figure 4), which we describe next.

The first part of the algorithm, lines 1–3, tests whether we are done with the variable scan (that is, the algorithm is called with the index $n + 1$) and, if so, prints μ. Now assume that $d \leq n$. So the currently iterated variable is x_d. We denote by v the owner of x_d, and by α the adhesion of v (as defined in Section 2). Moreover, we assume that the nodes owned by v are $x_l, \dots, x_k$ in ascending indices. Observe that $\mathrm{owned}(v)$ is indeed a consecutive set of variables, since the order is compatible with t.

In lines 8–15 we handle the case where we have just entered v from a different node of t, which means that x_d is the first node x_l owned by v, and v is not the root (that is, $v > 1$). From our construction, the adhesion of v is already assigned values in μ (again due to compatibility), and we check whether there is a cache hit for $\mu_{|\alpha}$ (the restriction of μ to α) in $cache_v$. If indeed there is a cache hit, then in lines 9–15 we scan the cache that contains all assignments μ' that we have already computed for $\mu_{|\alpha}$. For each such μ', we extend μ with μ' and adjust the trie structure $\mathcal{T}$ according to μ'. By *adjusting* $\mathcal{T}$ we consider every variable x_i in $x_l, \dots, x_k$ and if x_i is later used for a join, then we position the pointer precisely where it should have been if we scanned the trie and got to $\mu'(x_i)$;[2] and if x_i is not used for a future join, then we do nothing. As an example, in our running example (Figure 2), we ignore x_5 if we have a cached assignment for it.

Lines 16–22 are executed in the case where we have not just entered v, or we do but had a cache miss on line 8. In this case, we continue exactly as in RJoin (Figure 4), but we also test whether x_d is the last variable owned by v (i.e., x_d is x_k). If so, we either cache or do not cache the assignment $\mu_{|\mathrm{owned}(v)}$ based on the underlying caching policy for $\mu_{|\alpha}$. Observe that this action may lead to an eviction of a previously stored entry for some $\mu'_{|\alpha}$.

EXAMPLE 3.3. We will now show how the scenario of Example 3.1 is realized in the algorithm CacheTrieJoin, where we consider again our running example (Figure 2). The algorithm first calls CacheRJoin(1), and the execution is the same as in RJoin, all the way until we reach the call to CacheRJoin(3) where we have $\mu(x_1) = 1$ and $\mu(x_2) = 2$. Observe that $\mathrm{owner}(x_3) = v$, which is a non-root node, $\mathrm{owned}(v) = \{x_3, x_4\}$ (hence, $l = 3$ and $k = 4$). Also note that $\mathrm{adhesion}(v) = \{x_2\}$. The test of line 7 is true, but that of line 8 is false since $cache_v$ is empty at that point. So, we continue to lines 16–19 and apply the different assignments for x_3, starting with $x_3 \mapsto 1$. We then call CacheRJoin(4), where we find the assignment $x_4 \mapsto 2$. The test of line 20 is true, since x_4 is the last owned by v. Therefore, we may decide (based on the applied caching policy) to cache the entry $x_2 \mapsto 2$ in $cache_v$, and then we store there the assignment $(x_3, x_4) \mapsto (1, 2)$. We later store in that entry the assignment $(x_3, x_4) \mapsto (2, 2)$.

Later in the execution, we call CacheRJoin(3) when we have $\mu(x_1) = 2$ and $\mu(x_2) = 2$. We may then find out that in $cache_v$ we have cached the entry of $x_2 \mapsto 2$, and we simply use the two tuples μ' that maps (x_3, x_4) to $(1, 2)$ and to $(2, 2)$, as in lines 9–13. However, if there is a cache miss then we repeat the above first execution of CacheRJoin(3). □

The following theorem states the correctness of the algorithm CacheTrieJoin. The proof is by a fairly straightforward application of the basic separation properties of a tree decomposition.

THEOREM 3.4. *Let q be a CQ, $\langle x_1, \dots, x_n \rangle$ an ordering of* vars(Q) *and $\langle t, \chi \rangle$ a TD compatible with $\langle x_1, \dots, x_n \rangle$. Let D be a database, and $\mathcal{T}$ a trie structure for* TrieJoin. *Algorithm* CacheTrieJoin$(q, \langle x_1, \dots, x_n \rangle, \langle t, \chi \rangle, \mathcal{T})$ *prints $q(D)$.*

[2] Technically, this is done by storing the position with μ' in $cache_v$.

3.3 Counting

We now describe a variation of CacheTrieJoin (Figure 5) for *counting* the number of tuples in $q(D)$. The counting algorithm, which we refer to as CacheTJCount, is depicted in Figure 5. The input is the same as that of CacheTrieJoin, and the flow is very similar. There are, however, a few key differences, and our explanation (next) will focus on these.

The algorithm CacheTJCount uses some new global variables and data structures. The variable $total$ counts the joined tuples throughout the execution, and in the end stores the required number. For every non-root node v of t we have a counter $intrmd(v)$ that stores the intermediate count of the assignments to the variables owned by the nodes in $t_{|v}$ (i.e., the subtree of t that consists of v and all of its descendants), given the assignment to $\mathrm{adhesion}(v)$ in the current iteration. More precisely, let i be the maximal number such that x_i is in the adhesion of v, and consider a partial assignment μ that is nonnull on precisely $x_1, \dots, x_i$. In an iteration where μ is constructed, $intrmd(v)$ will eventually hold the number of assignments μ' that TrieJoin can assign to the variables owned by the nodes in $t_{|v}$. As $\langle t, \chi \rangle$ is compatible with the ordering $\langle x_1, \dots, x_n \rangle$, this number is the same for all assignments μ that agree on the adhesion α of v. The counter $intrmd(v)$ holds the correct value once we are done with the variables owned by v. Another fundamental difference from CacheTJCount is that now $cache_v$ stores a natural number (rather than a collection of assignments) for each assignment μ_α; this number is precisely the value of $intrmd(v)$ once we are done with the variables in $t_{|v}$.

Following the initialization, the algorithm calls the subroutine CacheRJoinCount, which is the counting version of CacheRJoin. The input takes not only the variable index d, but also a factor f that aggregates cached intermediate counts. When we are done scanning all of the variables (i.e., we reach line 2), the factor f is added to $total$. When we are at the first node owned by the current non-root owner v (lines 7–13), we reset the counter $intrmd(v)$. If we have a cache hit for $\mu_{|\alpha}$ in $cache_v$, then we copy the number $cache_v(\alpha)$ into $intrmd(v)$, multiply f by this number, and jump directly to the first index outside of $t_{|v}$ (line 12). This skipping is where compatibility is required, since it ensures that the nodes owned by $t_{|v}$ constitute a consecutive interval in $\langle 1, \dots, n \rangle$.

As previously, lines 14–20 are executed in the case where we have not just entered v, or experience a cache miss. We then continue as in RJoin. However, if x_d is the last variable owned by v, then we update the intermediate count by adding the product of the intermediate results $intrmd(c)$ of the children c of v. (Note that this product is 1 when v is a leaf.) Finally, in lines 22–23 we consider again the case where we have just entered a node v. Then, we are about to go back to the previous node, and so we apply the caching policy to possibly cache the number $intrmd(v)$ for $\mu_{|\alpha}$ in $cache_v$. (This is why we maintain $intrmd(v)$ to begin with.)

Algorithm CacheTJCount$(q, \langle x_1, \dots, x_n \rangle, \langle t, \chi \rangle, \mathcal{T})$

```
1: for $d = 1, \dots, n$ do
2:   $\mu(x_d) := \bot$
3: for all nodes $v$ of $t$ do
4:   $cache_v := \emptyset$
5:   $intrmd(v) := 0$
6: $total := 0$
7: CacheRJoinCount$(1, 1)$
8: return $total$
```

Subroutine CacheRJoinCount(d, f)

```
1: if $d = n + 1$ then
2:   $total := total + f$
3:   return
4: $v := \mathrm{owner}(x_d)$
5: $\{x_l, x_{l+1}, \dots, x_k\} := \mathrm{owned}(v)$
6: $\alpha := \mathrm{adhesion}(v)$
7: if $v \neq \mathrm{root}(t)$ and $d = l$ then
8:   $intrmd(v) := 0$
9:   if $\mu_{|\alpha}$ is a cache hit in $cache_v$ then
10:     $intrmd(v) := cache_v(\mu_{|\alpha})$
11:     $m := \max\{i \mid \mathrm{owner}(x_i) \text{ is in } t_{|v}\}$
12:     CacheRJoinCount$(m + 1, f \cdot cache_v(\mu_{|\alpha}))$
13:     return
14: for all matching values $a$ for $x_d$ in $\mathcal{T}$ do
15:   position $\mathcal{T}$ on $x_d \mapsto a$
16:   $\mu(x_d) := a$
17:   CacheRJoinCount$(d + 1, f)$
18:   if $v \neq \mathrm{root}(t)$ and $d = k$ then
19:     let $c_1, \dots, c_k$ be the children of $v$ in $t$
20:     $intrmd(v) := intrmd(v) + \prod_{i=1}^{k} intrmd(c_i)$
21: reset $x_d$ pointers in $\mathcal{T}$
22: if $v \neq \mathrm{root}(t)$ and $d = l$ then
23:   ApplyCachePolicy$(cache_v, \mu_{|\alpha}, intrmd(v))$
```

Figure 6: Cached count over trie join

EXAMPLE 3.5. We illustrate CacheTJCount on our running example (Figure 2). On CacheRJoinCount$(1, 1)$ we set (in lines 16–17) $\mu(x_1) = 1$ and call CacheRJoinCount$(2, 1)$, where we set $\mu(x_2) = 2$ and call CacheRJoinCount$(3, 1)$. We reach line 8 and initialize $intrmd(v)$ to 0. We have a cache miss (as the cache is empty), and we reach line 14, where CacheRJoinCount$(4, 1)$ is called with $\mu(x_3) = 1$. From there we call CacheRJoinCount$(5, 1)$ with $\mu(x_4) = 2$. Let c_l and c_r be the left and right children of v, respectively. When the call returns, we have $intrmd(c_l) = 2$ and $intrmd(c_r) = 2$, as x_5 can be mapped to 2 and 3 and x_6 can be mapped to 1 and 2. At this point $total$ is equal to 4, since the scan has ended four times. We then reach line 18 (since x_4 is the last owned by v) and set $intrmd(v) = 0 + 2 \times 2 = 4$. Similarly, after the call to CacheRJoinCount$(4, 1)$ with $\mu(x_3) = 2$ there will be a call with $\mu(x_4) = 1$ and $intrmd(v)$ will be incremented by another 4, and so will be the case with $\mu(x_4) = 2$. So, when we reach line 23 for $d = 3$, we may cache the number 12 as $cache_v(\mu_\alpha)$.

The next time CacheRJoinCount$(3, 1)$ is called with $\mu(x_2) = 2$ (i.e., when $\mu(x_1) = 2$), we check $cache_v$ and may find that $cache_v(\mu_\alpha)$ exists (i.e., cache hit) with $cache_v(\mu_\alpha) = 12$. If so, we reach line 12 and call CacheRJoinCount$(7, 1 \times 12)$. As $7 > n$, we skip to line 2 and add 12 to $total$. If there is a cache miss for μ_α in $cache_v$, there might still be a cache hit when we call CacheRJoinCount$(5, 1)$ with $\mu(x_3) = 2$, and then we immediately call CacheRJoinCount$(6, 1 \times 2)$ on line 12, as 6 is the minimal index outside the subtree of c_l (which contains only c_l). □

The following theorem states the correctness of the algorithm CacheTJCount.

THEOREM 3.6. *Let q be a CQ, $\langle x_1, \dots, x_n \rangle$ an ordering of* vars(Q) *and $\langle t, \chi \rangle$ a TD that is compatible with $\langle x_1, \dots, x_n \rangle$. Let D be a database, and $\mathcal{T}$ a trie structure for* TrieJoin. *Algorithm* CacheTJCount$(q, \langle x_1, \dots, x_n \rangle, \langle t, \chi \rangle, \mathcal{T})$ *computes $|q(D)|$.*

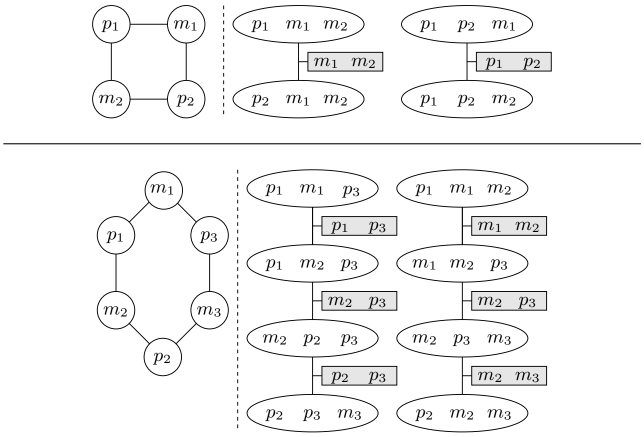


Figure 7: 4-cycle (top) and 6-cycle (bottom) queries on IMDB, each with two isomorphic TDs

Subroutine $\mathsf{GenericTD}(g, C)$

```
1:  $\langle S, U\rangle \leftarrow \mathsf{ConstrainedSep}(g, C)$
2:  if $S = \bot$ then
3:      return the singleton decomposition of $g$
4:  $\langle t_0, \chi_0\rangle := \mathsf{RecursiveTD}(g[S \cup U], C \cup S)$
5:  let $V_1, \dots, V_k$ be the connected comps. of $g - (S \cup U)$
6:  for $i = 1, \dots, k$ do
7:      $\langle t_i, \chi_i\rangle := \mathsf{RecursiveTD}(g[S \cup V_i], S)$
8:  let $t$ be obtained from $t_0, t_1, \dots, t_k$ by connecting the root
    of $t_0$ to the root of $t_i$ for all $i > 1$
9:  $\chi := \cup_{i=0}^{k} \chi_i$
10: return $(t, \chi)$
```

Figure 8: Tree decomposition via adhesion selection

The proof is more involved than Theorem 3.4, and has two steps. We first prove, by induction on time, that whenever we complete iterating the variables of v, the number $\mathit{intrmd}(v)$ is correct (i.e., it is the number of intermediate results for $t_{|v}$ given the assignment for $\mathrm{adhesion}(v)$). In the second step we show that every unit added to total accounts for a unique tuple in $q(D)$ and vice versa.

4. DECOMPOSITION

We now discuss the challenge of finding a TD $\langle t, \chi\rangle$ and a compatible variable ordering. A typical TD algorithm aims at optimising some specific cost function such as *generalized/fractional hypertree width* [12, 15]. In our case, an important factor in the effectiveness of the caches in our algorithms is their dimensionality, which is determined by the size of the adhesions. To better capture opportunities of a high skew and hit rate, we give precedence to keys from a domain of a smaller dimension, and hence, we favor smaller adhesions. There are, however, additional criteria beyond the topological properties of the TD. For example, we would like to use adhesions such that their corresponding subqueries have high *skews* in the data, and then caching a small number of intermediate results can save a lot of repeated computation. Moreover, we would like to have a TD that is compatible with an order that is estimated as good to begin with. For a (rather extreme) illustration, Figure 7 depicts two TDs of two queries, 4-cycle and 6-cycle, over the IMDB dataset (see Section 5), where m and p denote movie and person identifiers, respectively. The left TD favors persons for caching and the right favors movies for caching. While the decompositions are isomorphic, their performance of counting varies greatly: 4-cycle took around 40 seconds with the left TD, and around 4,000 with the right one; and 6-cycle took around 600 seconds and 27,000 on the left and right TDs, respectively.

We take the approach of generating many TDs, estimating a cost on each, and selecting the one with the best estimate. In our implementation (described in the next section), we deploy a heuristic cost function that ranks TDs based on three criteria, in a lexicographic manner: the maximal size over the adhesions (lower is better), the number of bags (higher is better), the sum of adhesions (lower is better), and the cost function of Chu et al. [10] for some variable ordering that is compatible with the TD.

4.1 Enumerating TDs

We now describe our technique for enumerating ordered TDs. In future work we plan to compare our enumeration to a recent one by Carmeli et al. [9]. We begin with a simple method for generating a single TD. The two common heuristics to generating TDs are *graph separation* and *elimination ordering* [7]. We adopt the former, as it will later allow us to plug in an algorithm for enumerating separating sets of a graph. The algorithm calls a method for solving the *side-constrained graph separation* problem, or just the *constrained separation* problem for short, which is defined as follows. The input consists of an undirected graph g and a set C of nodes of g. The goal is to find a *separating set* S of g, that is, a set S of nodes such that $g - S$ (obtained by removing from g every node of S) is disconnected. In addition, S is required to have the property that at least one connected component in $g - S$ is disjoint from C. Hence, S is required to separate C from some nonempty set of nodes. We call S a *C-constrained* separating set. We denote a call for a solver of this problem by $\mathsf{ConstrainedSep}(g, C)$. We later discuss an actual solver. For convenience, we assume that a solver returns the pair $\langle S, U\rangle$, where U is the set of all nodes in the connected components of $g - S$ that intersect with C.

The algorithm, called $\mathsf{GenericTD}(g, C)$, is depicted in Figure 8. It takes as input a graph g and a set C of nodes that is empty on the first call. The algorithm returns an ordered TD of g with the property that the root bag contains all nodes in C. So, the algorithm first calls $\mathsf{ConstrainedSep}(g, C)$. Let $\langle S, U\rangle$ be the result. It may be the case that the subroutine decides that no (good) C-constrained separating set exists, and then the returned S is null ($\bot$). In this case, the algorithm returns the TD that has only the nodes of g as the single bag. This case is handled in lines 1–3. Suppose now that the returned $\langle S, U\rangle$ is such that S is a C-constrained separating set. Denote by $V_1, \dots, V_k$ the connected components of $g - (S \cup U)$. The algorithm is then applied recursively to construct several ordered TDs. First, an ordered tree decomposition $\langle t_U, \chi_U\rangle$ of the induced subgraph of $S \cup U$, which we denote by $g[S \cup U]$, such that the root contains $C \cup S$ (line 4). Second, for $i = 1, \dots, k$, an ordered tree decomposition $\langle t_i, \chi_i\rangle$ of $g[S \cup V_i]$ (the induced graph of $S \cup V_i$) such that the root bag contains S (lines 5–7). Finally, in lines 8–10 the algorithm combines all of the tree decompositions into a single tree decomposition (returned as the result), by connecting the root of each $\langle t_i, \chi_i\rangle$ to $\langle t_U, \chi_U\rangle$ as a child of the root.

The algorithm $\mathsf{GenericTD}(g, C)$ of Figure 8 generates a single ordered TD. We transform it into an enumeration algorithm by replacing line 1 with a procedure that efficiently enumerates C-constrained separating sets, and then executing the algorithm on every such set. A key feature of the enumeration is that it is done by *increasing size* of the separating sets, and hence, if we stop the enumeration of separating sets after k sets have been generated (to bound the number of the generated TDs), *it is guaranteed that we have seen the k smallest C-constrained separating sets*.

We are then left with the task of enumerating the C-constrained separating sets by increasing size. For that, we have devised an algorithm that establishes the following complexity result.

THEOREM 4.1. *The S-constrained separating sets of a graph g can be generated by increasing size with polynomial delay.*

Our algorithm uses the well known technique for ranked enumeration with polynomial delay, namely the Lawler-Murty's procedure [17, 20][3] that reduces a general ranked (or *sorted*) enumeration problem to an optimization problem with simple constraints. Roughly speaking, to apply the procedure to a specific setting, one needs just to design an efficient solution to a *constrained* optimization problem. Due to lack of space, we omit the details and defer them to the long version of the paper.

5. EXPERIMENTAL STUDY

Our experimental study examines the performance benefits of our approach and algorithms. We compare our implemented algorithms to state-of-the-art solutions, and explore the effect of a number of key parameters and design choices.

5.1 Algorithms and Systems Evaluated

Our evaluation compares between implementations of several join algorithms, as listed below. All implementations were compiled using g++ 4.9.3 with the -O3 flag.

Our algorithms are CacheTrieJoin for CQ evaluation (Figure 5) and CacheTJCount for CQ counting (Figure 6). These implementations extend the vanilla implementation of LFTJ [29], which we describe below. We refer to the implementations by the acronyms CTJ-E and CTJ-C, respectively. The caches are implemented using STL's unordered_map. The computation of a TD is as described in Section 4. If no bound is mentioned for the cache size, then no eviction takes place (and every partial assignment is cached). We compare against the following alternatives.

LFTJ: We use a vanilla implementation of LFTJ [29]. Our implementation uses C++ STL *map* as the underlying Trie data structure. Notably, this implementation adheres to the complexity requirements of LFTJ.

YTD: This algorithm combines Yannakakis's acyclic join algorithm [30] with a TD, as described by Gottlob et al. [14]. The implementation is based on DunceCap [24]. For each intermediate join (bag) a worst-case optimal algorithm is used. The complexity requirement for the indices seekLowerBound is provided by a binary search, enabled through the use of the cascading vectors for the Trie. We use the query compiler from EmptyHeaded [1] (which uses a YTD-like algorithm) to generate the TD and variable ordering. For queries with only two bags we use a regular join since, in this case, the Yannakakis reduction stage generates an unnecessary overhead. Moreover, for count queries whose TDs yield more than two bags, we save the relevant result for the matching join attributes (rather than storing full intermediate results). Notably, we have experimented with alternative YTD implementations, but they all proved inferior to the one described above.

YTD-Par: EmptyHeaded [1] is a state-of-the-art graph query engine that operates as a parallel implementation of DunceCap [24], with optimizations for graph databases. We view it as a *query engine* rather than a pure algorithm, since the implementation is tied to the hardware: it parallelizes the execution through the Single-Instruction Multiple-Data (SIMD) model. Parallel operations are executed using the *vector unit* available on modern Intel processor cores. Specifically, *each core* on our test platform (Intel Xeon E5-2630 v3) includes a 256-bit vector unit that executes 8 integer (4-byte) operations in parallel.

Dataset	#Nodes	#Edges	Category
ca-GrQc	5,242	14,496	Collaboration net
p2p-Gnutella04	10,876	39,994	P2P net
ego-Facebook	4,039	88,234	Social net
wiki-Vote	7,115	103,689	Social net
ego-Twitter	81,306	1,768,149	Social net
imdb-Actresses	2,714,695	4,700,000	Movies
imdb-Actors	3,539,013	7,000,000	Movies

Table 1: Dataset (SNAP) statistics

In addition to pure algorithms, we also experiment with full systems. Pure algorithm implementations avoid the overhead associated with a full DBMS. We make this comparison simply to provide a context for the recorded running times.

LB-LFTJ: LogicBlox (LB) 4.3.18 [3]: A commercial DBMS configured to use LFTJ as an its join engine.

LB-FAQ: LogicBlox (LB) 4.3.18 configured to use InsideOut [16] as its join engine.

PGSQL: *PostgreSQL* [27] is an open-source relational DBMS (version 9.3.4). For query evaluation (as opposed to *count*), we use the *curser* API of PGSQL to avoid storage of join results in memory.

Other popular DBMSs and graph engines were compared to the above systems in a previous study [22], and were shown to be inferior in performance. Hence, we omit the other DBMSs from our experimental study. We further emphasize that our experiments explicitly restricted all algorithms and systems to utilize only a single core on the test machine, which does not affect YTD-Par SIMD parallelization.

5.2 Methodology

The setup and methodology we adopted in our experimental study are as follows.

Workloads. In par with other studies on join algorithms our evaluation is based, for the most part, on datasets from the SNAP collection [18], similarly to Nguyen et al. [22]. The datasets consist of wiki-Vote, p2p-Gnutella04, ca-GrQc, ego-Facebook and ego-Twitter. Table 1 gives some basic statistics on the datasets. As the distribution of values in SNAP dataset is highly skewed, we also use IMDB to explore the effect of datasets that are less skewed and whose data skew is not uniform across attributes. To this end, we partition IMDB's *cast_info* table into a *male_cast* and a *female_cast* tables, each with attributes *(person_id* and *movie_id)*. We exclude the TPC-C and TPC-H benchmarks as the join queries in these benchmarks are small.

Queries. Our datasets can be viewed as graphs, and so, we experiment using 3 types of CQs (again, consistently with Nguyen et al. [22]). The first type, denoted n-path for $n = 3, \dots, 7$, finds all paths of length n. For example, the 4-path CQ is

$$E(x,y), E(y,z), E(z,w) .$$

[3] Lawler-Murty's procedure is a generalization of Yen's algorithm [31] for finding the k shortest simple paths of a graph.

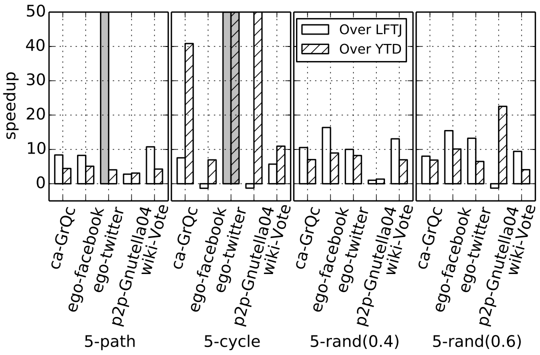


Figure 9: The speedups obtained with CTJ-E over LFTJ and YTD for full query evaluation. Bars that represent executions that timed out are marked as gray.

The second type is n-cycle, where $n = 3, \ldots, 6$, and the query finds cycles of length 3 to 6. For example, the 4-cycle CQ is

$$E(x,y), E(y,z), E(z,w), E(w,x) .$$

The third type consists of *random* CQs. We generate such CQs by forming a graph pattern using the Erdös-Reyni generator. The generator takes n nodes and adds an edge between every two nodes, independently, with a specified probability p. The graph is undirected and without self loops. We use only connected graphs with $n = 5$ and $n = 6$, and with $p = 0.4$ and $p = 0.6$. Random graph queries are denoted as $n\text{-rand}(p)$. For each set of parameters we generate four different graphs. We do not examine *clique* queries as these cannot be decomposed, and hence our algorithms are the same as LFTJ in this case.

Hardware and system setup. Our experimental platform uses Supermicro 2028R-E1CR24N servers. Each server is configured with two Intel Xeon E5-2630 v3 processors running at 2.4 GHz, 64GB of DDR3 DRAM, and is running a stock Ubuntu 14.0.4 Linux.

Testing protocol. Each experiment was run three times, and the average runtime is reported. We set an execution timeout of 10 hours. Executions that timed out are highlighted.

5.3 Experimental Results

We start by experimenting with unlimited caches on query evaluation of CQs. Next, we compare different cache sizes, caching policies and other cache attributes.

Query evaluation produces all the tuples in the result of the query. We focus our exploration of query evaluation on *computing* the materialized result rather than *storing* it. With the help of the related parties, the algorithms and systems were configured to ignore the final result and not store it. The only exception is YTD-Par, for which we could not disable the materialization of the final result. It is therefore not shown in our examination of query evaluation.

Figure 9 presents the results for running query evaluation of 5-path and 5-cycles queries. The figure shows that for 5-path queries, CTJ-E outperforms YTD by 4× and LFTJ by over 9×. The performance gap is attributed to CTJ-E's caching, which captures frequently used intermediate results. CTJ-E's caching eliminates redundant scans of the Trie structure that occur in LFTJ. CTJ-E also outperforms YTD by up to 4.6× (3.2× on average), because the computation of YTD becomes memory bound in the final join stages due to the memory complexity of the Yannakakis joins.

For 5-cycle queries, Figure 9 shows that CTJ-E is faster than LFTJ by an average of 8× for ca-GrQc, twitter and wiki datasets.

Query	Algorithms			Systems		
	CTJ-E	YTD	LFTJ	LB-FAQ	LB-LFTJ	PGSQL
3-path	23	2×	1.3×	46×	34×	10116×
4-path	133	4×	5×	17×	26×	27679×
5-path	2222	4×	8×	23×	19×	t/o
6-path	78528	4×	10×	23×	23×	t/o
7-path	3265542	4×	10×	t/o	t/o	t/o
4-cycle	558	1.1×	1.9×	9×	5×	4409×
5-cycle	4125	41×	8×	11×	19×	11526×
6-cycle	84248	9×	14×	40×	37×	564×

Query	Algorithms			Systems		
	CTJ-E	YTD	LFTJ	LB-FAQ	LB-LFTJ	PGSQL
3-path	72	1.5×	3×	14×	17×	18355×
4-path	791	4×	9×	23×	21×	59157×
5-path	29458	4×	11×	26×	24×	t/o
6-path	1.33e+06	4×	11×	26×	t/o	t/o
7-path	t/o	t/o	t/o	t/o	t/o	t/o
4-cycle	2192	0.7×	1.7×	5×	4×	2312×
5-cycle	27855	11×	6×	3×	14×	t/o
6-cycle	415783	4×	17×	5×	44×	t/o

Figure 10: CTJ-E runtimes (in msecs) for {3–7}-path and {4–6}-cycle queries and relative runtimes for compared solutions (i.e., m× means m times slower than CTJ-E), for ca-GrQc (top) and Wiki (bottom) datasets. Timeout (t/o) means over 10 hours. YTD-Par is omitted from the comparison as it always stores the materialized result.

For the facebook and p2p-Gnutella04 datasets, however, CTJ-E experiences a small slowdown. Finally, CTJ-E outperforms YTD by 26× on average. The reason is that YTD's Yannakakis and the worst-case optimal join algorithm used by YTD, favor the opposite attributes order, which dramatically affects its performance.

CTJ-E also delivers performance benefits for sparse random pattern queries. Figure 9 shows the results for representative graphs (which are consistent with the results for the other graphs). Specifically, for $5\text{-rand}(0.4)$ queries, CTJ-E outperforms LFTJ by 5× on average. CTJ-E is also consistently 3–4× faster than YTD, with the exception of p2p-Gnutella04 for which the results are comparable. These trends are consistent for denser $5\text{-rand}(0.6)$ random graphs. Here too, the results demonstrate the effectiveness of CTJ-E, whose runtime is, on average, 10× faster than LFTJ and 7× than YTD (CTJ-E and LFTJ runtimes are comparable for p2p-Gnutella04).

Figure 10 presents the results of query evaluation for {3–7}-path and {4–6}-cycle queries over the ca-GrQc (top) and Wiki (bottom) datasets for different algorithms and systems. For brevity, we show the results for only two datasets: ca-GrQc and Wiki. These results are consistent with the results obtained for the other SNAP datasets.

The figure shows that the performance benefits of CTJ-E over LFTJ increase with the size of the query. CTJ-E is 10× faster than LFTJ for 7-path queries and 14× for 6-cycle queries. Compared to YTD, CTJ-E speedup is 4× for 7-path queries and 4–9× for 6-cycle queries. The only case where CTJ-E is slower than another algorithm is the small 4-cycle query, for which YTD is faster by 30% on the Wiki dataset. Importantly, these results are consistent across the other datasets, excluding p2p-Gnutella04 for which the algorithms are comparable.

Figure 10 also compares the performance of our algorithms to that of full DBMSs (PGSQL, LB-LFTJ, and LB-FAQ). We observe that the speedups are even larger (as expected, due to system overhead). Notably, the ratio between the performance of LFTJ and LB-LFTJ is more or less constant, showing that the system overhead here accounts for around 2× slowdown.

dataset	5-path 25%	10%	1%	5-cycle 25%	10%	1%
ca-GrQc	1.7×	3×	18×	1.9×	3×	5×
p2p-Gnutella04	4×	5×	6×	1.3×	1.3×	1.3×
ego-twitter	6×	9×	18×	2×	2×	2×
wiki-Vote	1.2×	1.3×	3×	4×	4×	4×

Table 2: Slowdown due to cache sizes with LRU, over unlimited cache size for 5-path and 5-cycle query evaluation

On average, CTJ-E is over 20× faster than LB-FAQ and LB-LFTJ for all path queries, and 3-44× faster for all cycle queries. An even more extreme speedup is evident when comparing to PGSQL, where CTJ-E is consistently 3–5 orders of magnitude faster.

To conclude, we have shown that CTJ-E is substantially faster than the alternatives. Furthermore, the performance benefits of CTJ-E increase with the size of the query.

5.3.1 Cache Parameters

Tuning the parameters of the CTJ-E cache (e.g., cache size, eviction policy, cache partitioning) do not affect the correctness of the CTJ-E. Instead, these parameters only affect the caching efficiency of CTJ-E and, by proxy, the performance of the algorithm. The caching of partial results in CTJ-E thus presents a tradeoff between memory consumption and performance. Interestingly, LFTJ and YTD represent the two extremes of this classic tradeoff. On one hand, LFTJ caches no partial or intermediate results but rather repeatedly scans the Trie to regenerate partial results. On the other hand, YTD must maintain all intermediate results generated by the individual joins on each bag. As a result, its memory consumption is even higher than CTJ-E with an unbounded cache.

In this section we explore the memory-performance tradeoff by examining the impact of the different cache parameters on the performance of LFTJ. Unless stated otherwise, the memory allocated for the cache is evenly partitioned across the individual caches.

Cache size. The size of the CTJ-E cache is, naturally, the primary parameter that affects caching performance. We explore this parameter's impact on performance by bounding the cache size to 1%, 10%, and 25% of the size needed to store all partial results. For example, a 5-cycle query running on the twitter dataset requires 476MB to cache all partial results. We thus examine CTJ-E performance when bounding the total cache size to 4.76MB (1%), 47.6MB (10%) and 119MB (25%). In this experiment, all bounded caches use the least-recently-used (LRU) eviction policy.

Table 2 presents the performance of CTJ-E with bounded cache size for representative queries and datasets. The performance is presented as the slowdown over a run with an unbounded cache.

As expected, the table shows that performance degrades when reducing the cache size. For example, the performance of a 5-path query on the ca-GrQc dataset (0.4MB unbounded cache) slows down by 1.7× when bounding the cache to 25% of full capacity, by 2.5× with 10% of full capacity, and by 18.4× with 1% of full capacity. The performance degradation is less acute in other cases. A 5-cycle query running on ca-GrQc (8.8MB unbounded cache) slows down by 1.9×, 3×, and 5× for caches bounded at 25%, 10%, and 1%, respectively, of full capacity. In other cases, the performance impact of a bounded cache is fairly constant regardless of the bound. For example, running a 5-cycle query on the twitter dataset (476MB unbounded cache) results in a slowdown of 2.4–2.5× for caches bounded at 1–25% of full capacity.

dataset	5-path	5-cycle
ego-twitter	1.8×	11×
ca-GrQc	1.3×	2×
p2p-Gnutella04	1.3×	-1.3×
wiki-Vote	-1.1×	4×

Table 3: Speedup for LRU over RANDOM on cache bounded to 10% for 5-path and 5-cycle query evaluation

dataset	5-path 1x	2x	5x	5-cycle 1x	2x	5x
ca-GrQc	11.6×	3.4×	3×	10.4×	9.2×	8.3×
wiki-Vote	2.3×	1.5×	1.6×	5.5×	5×	5×
p2p-Gnutella04	5×	5×	5×	1.3×	1.3×	1.3×
ego-twitter	14×	8×	5.5×	2.4×	2.4×	2.4×

Table 4: Slowdown due to different cache partitioning with LRU bounded to 5% of the full cache capacity, over unlimited cache size for 5-path and 5-cycle query evaluation)

In summary, Table 2 shows that bounding the cache size to 25% of its full capacity only yields an average slowdown of ∼3× over an unbounded CTJ-E run. Bounding the cache size even further to 10% of an unbounded cache results in an average slowdown of ∼2.7× over the performance obtained with an unbounded cache. Notably, the performance obtained with a 10% bound is still superior to LFTJ, as well as to YTD 5-cycle queries, and comparable to YTD for 5-path queries.

Eviction policy. We now turn to examine the impact of the cache eviction policy on overall CTJ-E performance. Specifically, we compare the performance obtained with both *LRU* and *Random* eviction policies (the Random policy, as its name suggests, randomly selects a cache entry to evict with uniform distribution). We note that we have experimented with other eviction and insertion policies, some based on statistical analysis of the datasets, but none provided much better results than the classic LRU policy.

Table 3 presents the LRU performance as speedup over Random. For brevity, we only show results for a 10% cache bound. The table shows that for path queries LRU outperforms Random by 1.4× on average. This is because most cached values will be effective in path queries on the datasets we tested, and due to the overhead of the LRU bookkeeping. On cycle queries, LRU outperforms Random by 4.5× on average. We therefore choose to use the LRU eviction policy with bounded CTJ-E caches.

Cache partitioning. The final parameter we explore is the allocation of memory among caches. We test the LRU performance speedup for bounded cache size, which is partitioned between the caches in three different configurations. The first configuration (1×) divides the allocated memory equally between the caches. The second (2×), divides the allocated memory between the caches, such that each level is bounded to 2× of the size of the level above it. Here, a cache *level* means the position in the pre-order of the TD. As an example, for the 5-cycle query on the twitter dataset, we allocate a total of 476MB. In the second configuration, the first cache will be bounded to 1/3 (158MB) and the second cache will be bounded to 2/3 (317MB). The last configuration (5×) is similar to the previous, but with a scale of 5× instead of 2×.

Table 4 shows the results for CTJ-E with LRU eviction, bounded to 5%, over the different partitioning configurations. The results show that for small caches of equal size, the caches can become

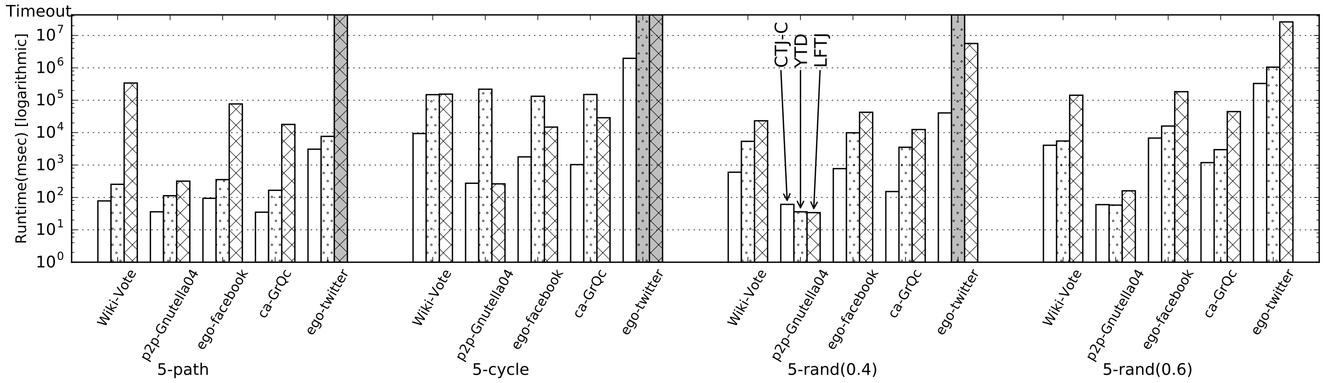


Figure 11: Runtimes for count queries using the different algorithms. Gray bars represent executions that timed out.

ineffective due to thrashing. The results also show that a different cache partitioning that allocates more memory to greater level caches, such as 2× and 5×, can improve the performance by 10%-3×. With these configurations CTJ-E outperforms LFTJ even with very small memory allocation. The reason we observed is that a cache in a greater level is accessed more often, and therefore accounts for a larger portion of the recurring joins. Note that different cache partitions do not affect queries on p2p-Gnutella04, since CTJ-E caches are less effective for this dataset. This crude allocation depicts the importance of dynamically allocating the memory between the caches, which we plan to pursue in future research.

Summary. We conclude that bounded caches enable CTJ-E to benefit from both worlds. On one hand, it delivers substantial speedups over LFTJ while preserving the bounded memory footprint property. On the other hand, it can execute in settings where traditional join algorithms, which store all intermediate results, either cannot execute or suffer substantial slowdowns due to disk I/O.

5.4 Results on Count Queries

We now examine the performance benefits of CTJ-C for count queries. Figure 11 presents the runtime of 5-path, 5-cycle, and 5-rand queries on different datasets. It shows that CTJ-C executes the queries substantially faster than the alternatives for all datasets except p2p-Gnutella04. CTJ-C is faster than LFTJ by over an order of magnitude. When compared to YTD, CTJ-C is typically 2–5× faster, with the exception of 5-rand(0.4) over p2p-Gnutella04, where CTJ-C results in a marginal slowdown.

The distinction between the datasets is rooted in their value distribution. Skewed value distributions are more amenable to caching. Specifically, when some values appear frequently in multiple tuples, caching partial walks through the LFTJ Trie will likely prevent redundant walks over the Trie. For example, the ego-Twitter dataset exhibits such skew. For this dataset, CTJ-C is consistently 2–5× faster than YTD and orders of magnitude faster than LFTJ.

On the other hand, when the distribution of values across the dataset is not skewed, as is the case with p2p-Gnutella04, caching partial values have little benefit. Indeed, for this dataset the performance benefits of CTJ-C are moderate (for 5-rand queries, both YTD and LFTJ even marginally outperform CTJ-C). The results demonstrate the effectiveness of CTJ-C when running on datasets whose value distribution is skewed.

Figure 11 compares the algorithms when running two representative 5-rand random graph queries. Comparing CTJ-C with the LFTJ algorithm, we see that CTJ-C is consistently faster by orders of magnitude. The only exception is the p2p-Gnutella04 that, as discussed above, exhibits a balanced value distribution. When comparing CTJ-C to YTD, we observe an average speedup of ~8×. Again, the only exception is the p2p-Gnutella04 dataset. Notably, the results for 6-rand (not shown) are consistent with 5-rand.

The performance benefits of CTJ-C are consistent across different query sizes. Figure 12 presents the runtimes for {3–7}-path and {3–6}-cycle queries. For brevity, we show the results for only two of the datasets. (The figure also shows the performance of DBMSs, which is discussed below.) The figure shows that for path queries CTJ-C is consistently 3× faster than YTD. Moreover, CTJ-C is orders of magnitude faster than LFTJ, and the performance benefits only increase with the size of the query.

For {3–7}-cycle queries, Figure 12 shows that CTJ-C outperforms LFTJ and YTD, especially on larger cycle queries. Interestingly, we see little difference in the running times for 3-cycle queries. The reason for that is there is no tree decomposition for triangles, and CTJ-C effectively behaves like LFTJ. Similarly, the performance of CTJ-C and YTD is comparable for 3-cycle queries.

When comparing the benefits of CTJ-C over large cycle and path queries (Figure 12), we see that CTJ-C delivers better speedups for paths. This is attributed to the cache dimension property (the size of adhesions). Therefore, the cache dimension for paths is set to one, and for cycles it is set to two. Notably, a cache whose dimension is one is shown to be much more effective. 5-cycle queries present another interesting result. For these queries YTD performs worse than LFTJ (and CTJ-C). The reason is that YTD's Yannakakis and the worst-case optimal join algorithm used by YTD, favor the opposite attributes order, which dramatically affects its performance.

Figure 12 shows that the performance benefit of CTJ-C and YTD over LFTJ increase with the query size at an exponential rate. Moreover, while CTJ-C and YTD have similar scaling trends for path queries, CTJ-C is an order of magnitude faster for {5–6}-cycle.

Comparison to systems and engines. To explore the scaling trends of the pure algorithms compared to those of DBMSs, we ran the queries on PGSQL (using pairwise join), LB-LFTJ, LB-FAQ (worst-case optimal join algorithms) and YTD-Par (parallel implementation of YTD). For brevity, we show the results for only two datasets: Wiki-Vote and ego-Facebook. Notably, these are consistent with the results obtained for the other SNAP datasets.

Figure 12 shows the results for {3–7}-path count queries. The first thing to note in the table is that the scaling of vanilla LFTJ and LB-LFTJ are correlated. We attribute the 4–10× ratio in performance between the two to overheads associated with running a full DBMS vs. a pure algorithm. A comparison between YTD-Par and YTD shows that YTD-Par is much faster than YTD. This to be expected, as YTD-Par engine is a parallel implementation of YTD pure algorithm, using the processor's wide vector unit. Due to the

	Algorithms			Systems and engines			
query	CTJ-C	YTD	LFTJ	LB-FAQ	LB-LFTJ	PGSQL	YTD-Par
3-path	36	3×	5×	9×	54×	19×	0.08×
4-path	58	3×	133×	5×	615×	364×	0.05×
5-path	78	3×	4362×	8×	21113×	11161×	0.09×
6-path	97	3×	157691×	7×	t/o	402735×	0.10×
7-path	119	4×	t/o	7×	t/o	t/o	0.13×
3-cycle	24	1×	1×	13×	13×	41×	0.21×
4-cycle	1474	0.85×	3×	7×	7×	3×	0.16×
5-cycle	9401	16×	16×	1.66×	43×	13×	2×
6-cycle	28615	11×	235×	1×	617×	242×	0.90×

	Algorithms			Systems and engines			
query	CTJ-C	YTD	LFTJ	LB-FAQ	LB-LFTJ	PGSQL	YTD-Par
3-path	26	5×	4×	14×	39×	22×	0.12×
4-path	48	4×	62×	10×	308×	174×	0.06×
5-path	94	4×	818×	7×	7973×	2116×	0.07×
6-path	119	3×	15086×	6×	t/o	56534×	0.08×
7-path	150	3×	t/o	6×	t/o	t/o	0.09×
3-cycle	48	1.1×	1×	12×	12×	42×	0.13×
4-cycle	569	1.31×	1×	5×	5×	4×	0.12×
5-cycle	1785	74×	8×	3×	37×	26×	12×
6-cycle	4639	54×	81×	3×	366×	208×	5×

Figure 12: CTJ-C runtimes (in msecs) for {3–7}-path and {3–6}-cycle count queries and relative runtimes for compared solutions (i.e., $m\times$ means m times slower than CTJ-C), shown for Wiki (top) and Facebook (bottom) datasets. *t/o* indicates runtime over 10 hours (timeout).

parallel implementation, YTD-Par is also faster than CTJ-C and LFTJ on path queries. Nevertheless, the sequential CTJ-C implementation is comparable to YTD-Par for {5–6}-cycles queries (and is even faster on some datasets).

On average, CTJ-C is over 39× faster than LB-LFTJ for all path queries, and 5-208× faster for all cycle queries. CTJ-C speedup over LB-FAQ is 7× and 4× on average for path and cycle queries, respectively. Compared to PGSQL, CTJ-C is consistently 3–5 orders of magnitude faster for big cycle and path queries.

6. CONCLUDING REMARKS

We have studied the incorporation of caching in LFTJ by tying an ordered tree decomposition to the variable ordering. The resulting scheme retains the inherent advantages of LFTJ (worst case optimality, low memory footprint), but allows it to accelerate performance based on whatever memory it decides to (dynamically) allocate. Our experimental study shows that the result is consistently faster than LFTJ, by orders of magnitude on large queries, and usually faster than other state of the art join algorithms.

This work gives rise to several directions for future work. These include further exploration of different caching strategies, different TD enumerations and cost functions, extension to general aggregate operators (e.g., based on the work of Joglekar et al. [15] and Khamis et al. [16]), and generalizing beyond joins [29]. A highly relevant work is that on *factorized representations* [6,23,25], which we can incorporate in two manners. First, our caches can hold factorized representations instead of flat tuples. Second, the final result can be factorized by itself, and in that case our caching is likely to become even more effective, since it will save the cycles then we effectively spend on de-factorizing our cached results.

Acknowledgments. We are very grateful to LogicBlox for sharing their code, and especially to Hung Ngo and Martin Bravenboer from LogicBlox for helpful suggestions and help with their system. We are also grateful to Christopher Aberger and Chris Ré from Stanford University for insightful discussions and assistance in setting up EmptyHeaded. This research is supported by the Israeli Ministry of Science, Technology, and Space. Yoav Etsion is supported by the Center for Computer Engineering at Technion. Benny Kimelfeld is a Taub Fellow supported by the Taub Foundation, and supported by the Israeli Science Foundation, Grants #1295/15 and #1308/15.

7. REFERENCES

[1] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *SIGMOD*, pages 431–446, 2016.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[3] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the logicblox system. In *SIGMOD*, pages 1371–1382, 2015.

[4] S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Alg. Disc. Meth.*, 8(2):277–284, 1987.

[5] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.

[6] N. Bakibayev, T. Kociský, D. Olteanu, and J. Zavodny. Aggregation and ordering in factorised databases. *PVLDB*, 6(14):1990–2001, 2013.

[7] H. L. Bodlaender and A. M. C. A. Koster. Treewidth computations i. upper bounds. *Inf. Comput.*, 208(3):259–275, 2010.

[8] V. Bouchitté, D. Kratsch, H. Müller, and I. Todinca. On treewidth approximations. *Discrete Applied Mathematics*, 136(2-3):183–196, 2004.

[9] N. Carmeli, B. Kenig, and B. Kimelfeld. On the enumeration of all minimal triangulations. *CoRR*, abs/1604.02833, 2016.

[10] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, pages 63–78, 2015.

[11] D. G. Feitelson. Metrics for mass-count disparity. In *MASCOTS*, pages 61–68, 2006.

[12] G. Gottlob, G. Greco, and F. Scarcello. Pure nash equilibria: Hard and easy games. *J. Artif. Intell. Res. (JAIR)*, 24:357–406, 2005.

[13] G. Gottlob, M. Grohe, N. Musliu, M. Samer, and F. Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In *WG*, pages 1–15, 2005.

[14] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. In *PODS*, pages 21–32, 1999.

[15] M. R. Joglekar, R. Puttagunta, and C. Ré. AJAR: aggregations and joins over annotated relations. In *PODS*, pages 91–106, 2016.

[16] M. A. Khamis, H. Q. Ngo, and A. Rudra. FAQ: questions asked frequently. In *PODS*, pages 13–28, 2016.

[17] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972.

[18] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection, 2014.

[19] D. Marx. Approximating fractional hypertree width. *ACM Trans. on Algorithms*, 6(2), 2010.

[20] K. G. Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, 16(3):682–687, 1968.

[21] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms: [extended abstract]. In *PODS*, pages 37–48, 2012.

[22] D. T. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra. Join processing for graph patterns: An old dog with new tricks. In *GRADES*, pages 2:1–2:8, 2015.

[23] D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. *ACM Trans. on Database Systems (TODS)*, 40(1):2, 2015.

[24] A. Perelman and C. Ré. DunceCap: Compiling worst-case optimal query plans. In *SIGMOD*, pages 2075–2076, 2015.

[25] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *SIGMOD*, pages 3–18, 2016.

[26] 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*, pages 23–34, 1979.

[27] The PostgreSQL Global Development Group. PostgreSQL. www.postgresql.org.

[28] S. Tu and C. Ré. DunceCap: Query plans using generalized hypertree decompositions. In *SIGMOD*, pages 2077–2078, 2015.

[29] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, pages 96–106, 2014.

[30] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.

[31] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17:712–716, 1971.

Querying Improvement Strategies

Guolei Yang
Department of Computer Science
Iowa State University
Ames, IA, USA
yanggl@iastate.edu

Ying Cai
Department of Computer Science
Iowa State University
Ames, IA, USA
yingcai@iastate.edu

ABSTRACT

Given a set of objects and a set of top-k queries on these objects, we are interested in adjusting some object's attribute values to meet some requirements under certain cost constraints. We call such an adjustment an *improvement strategy*. Searching for cost-efficient improvement strategies is crucial for applications like product marketing, where top-k queries are used to model users' preference. We propose two types of *Improvement Queries (IQs)*. A *Min-Cost* IQ finds the improvement strategy that makes selected objects hit a desired number of queries with the minimal cost, while a *Max-Hit* IQ searches for the improvement strategy that makes selected objects hit as many queries as possible with a given budget. We show that answering IQs is NP-hard and develop a suite of heuristic algorithms. Our key idea is to interpret objects as functions and treat each top-k query as an input to the functions. The geometric relationship among the function intersections is then leveraged for efficient query processing. We implement the proposed algorithms as an analytic tool and integrate it with a DBMS, and they exhibit excellent performance on both synthetic and real-world data in experiments.

1. INTRODUCTION

Top-k query [7, 6, 11, 26] is widely used in applications like e-commerce for users to find objects (e.g., products) that best match their preference. A user's preference is represented by a *utility function* which computes a "score" for each object, and a top-k query retrieves the k objects with the highest/lowest scores. When an object appears in a query result, we say the object *hits* the query. Given a set of objects and a set of top-k queries, adjusting an object's attribute values could result in changing the number of queries it hits. In this paper, we refer to such an adjustment as an *improvement strategy*. We are interested in querying the improvement strategies for objects of interested under some cost constraint. We consider two variations of *Improvement Query* (IQ):

- **Min-Cost IQ:** Given a *cost function*, this type of IQ finds the most cost-efficient improvement strategy for an object to hit a given minimum number of top-k queries. Here a cost function is defined by the query issuer to measure the cost of adjusting attribute values of objects. The idea of modeling costs as math functions is a common approach [19, 4]. We allow query issuers to define their own cost functions.

- **Max-Hit IQ:** Given a *cost function* and a *budget*, this type of IQ returns the improvement strategy for an object to hit the maximal number of top-k queries under the condition that the total cost does not exceed the budget.

The problem of finding improvement strategies arises from a variety of applications. For example, a camera manufacturer may want to improve its product for more market shares. Here an improvement is a change of the product's features such as camera's resolution and price. Likewise, in a presidential election, it is imperative for the candidates to evaluate their campaign strategies from time to time, and adjust if needed, in order to appeal themselves to more voters. In these examples, there are a set of objects (e.g., products, presidential candidates) and a set of top-k queries, each representing the preference of a user (e.g., customer, voter), and we want to improve one or more objects (called *targets*) to hit as many queries as possible. Existing queries such as reverse top-k query [21], maximal rank query [14], and reverse k-ranks query [25] have been developed to provide information concerning an object's competitiveness in top-k selection. These queries, however, do not allow one to identify an improvement strategy, the focus of this paper.

The problem of processing IQs can be formulated as constrained optimization problems and we prove it is NP-hard. As such, finding accurate query results is computation intensive even for moderate size datasets. We address this problem by proposing a suite of heuristic algorithms. At the core of the proposed algorithms is a novel indexing technique. Our key idea is to 1) *interpret each object as a function*, and 2) *treat each top-k query as an input to these functions.* The *intersection* of two functions formulates a hyperplane in their domain. Given a set of functions, their intersection hyperplanes partition the domain into a number of *subdomains.* We observe that the rank of an object must be the same for all queries that fall in one subdomain. Applying an improvement strategy to an object will cause the boundary of some subdomains to change, but it will affect the result of a top-k query only if the query falls into a different subdomain. This observation allows us to develop a highly

Series ISSN: 2367-2005

10.5441/002/edbt.2017.27

efficient algorithm for IQ processing. We summarize our main contributions as follows:

- To our knowledge, this is the first to study the problem of object improvement, defined as adjusting the attribute values of the objects of interest. We prove the inherent intractability of the minimal cost/maximal hit improvement strategy searching problem.

- We propose the notion of *Improvement Query (IQ)*, which supplements the existing top-k query with the key information needed to develop effective improvement strategies. We propose two types of IQs: Min-Cost IQ and Max-Hit IQ. Given a user-defined cost function, the former IQ finds the most cost-efficient improvement strategy that achieves desired number of hits, while the latter one finds the improvement strategy that hits the maximal number of top-k queries with a given budget. We design efficient IQ processing algorithms based on a novel query indexing technique and an important observation.

- We implement the proposed techniques as an analytic tool and integrate it with the Database Management System (DBMS). The tool is thoroughly evaluated over synthetic and real-world data. The results show that our techniques demonstrate good performance, and the tool is scalable for large-scale users and objects.

The rest of the paper is organized as follows. We discuss related work in Section 2. In Section 3, we formally define the problem and give an overview of our solution. The proposed techniques for basic cases and complex scenarios are presented in Section 4 and Section 5 respectively. In Section 6, we describe our system implementation and present experiment results. We conclude the paper in Section 7.

2. RELATED WORK

Our work is closely related to the top-k query and other rank-aware queries. We briefly discuss some representative works as follows.

Top-k query: Several indexing techniques have been proposed for efficient processing of top-k queries. View-based techniques (e.g., [11, 8]) employs materialized views to retrieve top k, where objects are ranked according to arbitrary utility functions. Layer-based technique ([6]) computes the convex hulls of data points, and organizes them in layers. Top-k queries are then processed from the outmost layer, which contains objects that are most likely to be in top-k. The state-of-the-art technique is [26], which exploits the dominant relationship between objects. More specifically, an object p_i is said to *dominate* another object p_j if there exists no linear utility functions that ranks p_i lower than p_j. Thus there is no way for p_j to be include in a query result unless p_i is included first. As such, objects can be organized into groups based on their dominant relationship. This allows efficient processing of top-k queries. These techniques, however, are all limited to linear utility functions. The problem of non-linear utility function top-k selection is studied in [24] as k-constrained optimization problem, and addressed with a state-space indexing technique.

Other rank-aware queries: Top-k query has inspired a rich family of rank-aware queries, which are closely related to our research. Given a set of objects and a set of top-k queries, a reverse top-k query [20, 21] retrieves the queries whose result contains a selected object. For less popular objects, a useful variant of reverse top-k query is the reverse k-ranks query, which can find the k queries whose rank of an object is the highest among all queries. A maximum rank query [14] computes the highest possible rank an object can achieve for any utility function. Unlike our work, the maximum rank is not achieved by adjusting attributes of the object itself, but by exploring different utility functions. It can find the maximum rank one object can get with respect to any query, but cannot provide information on how to increase the number of queries that an object hits. This makes it fundamentally different from our problem. These existing queries help one understand the current competitiveness of an object among its peers, but not improve the object to make it more competitive.

Another related work is [13]. It considers how to find the k objects from a dataset that can be upgraded with minimal cost. The goal of *upgrade* is to make the object appear on *skyline* of the dataset. An object is said to be on skyline if it is not worse in all dimensions than another object in the dataset. Each dimension is compared independently and no function is computed, therefore making an object to be on skyline is straightforward, and the major challenge addressed in [13] is how to efficiently find the k objects with lowest upgrading cost without traversing all objects. In contrast, finding optimal improvement strategy for even one object is NP-hard. Their proposed algorithm cannot solve our problem. A similar work [22] discusses how to efficiently create new products that appear on skyline of a given dataset. But it does not consider improving existing objects, thus less related to our work.

3. PRELIMINARIES

3.1 Problem Definition

Consider a dataset D with n objects. Each object p_i is a point in the d-dimensional space, where each dimension represents a numerical attribute of the object. We use $p_i^{(j)}$ to denote its j-th dimension's value. Each dimension can be continuous or discrete, finite or infinite. Let $Q = \{q_1, q_2, ..., q_m\}$ denote a set of m top-k queries. Each query q_i $(1 \leq i \leq m)$ specifies a k value (i.e., the number of object to return) and a utility function which computes a score for each object. Together they represent a user's preference. The number of top-k queries hit by p_i is denoted by $H(p_i)$. We define improvement strategy as follows:

DEFINITION 1 (IMPROVEMENT STRATEGY). *An improvement strategy s for an object p_i is a d-dimensional vector $s = \{s_1, s_2, ..., s_d\}$, where $s_i \in \mathbb{R}$ specifies how the i-th attribute is to be adjusted, i.e., applying s to p_i will replace p_i with a new object p_i', where $p_i'^{(j)} = p_i^{(j)} + s_j$ $(1 \leq j \leq d)$.*

To illustrate, consider a camera dataset showed in Figure 1. Each camera has three discrete attributes *resolution*, *storage*, and *price*. Together they determine the camera's rank for a given top-k query. Let $s = \{5, 2, -50\}$ be an improvement strategy. Applying s on a camera means to increase the camera's resolution by 5 Megapixel, increase its storage by 2 GB, and decrease its price by \$50. For example, applying s on camera p_1 will result in a new object $p_1' = \{15, 4, 200\}$. Note that after the improvement, p_1''s

Cameras

ID	*resolution (Megapixel)*	*storage (GB)*	*price ($)*
p_1	10	2	250
p_2	12	4	340
$\cdots$	$\cdots$	$\cdots$	$\cdots$

$\Downarrow$ *Applying* $s = \{5, 2, -50\}$ *to* p_1

ID	*resolution (Megapixel)*	*storage (GB)*	*price ($)*
p'_1	15	4	200
p_2	12	4	340
$\cdots$	$\cdots$	$\cdots$	$\cdots$

Top-k queries represent users' preference for camera

ID	*Utility function*	*top-k*
q_1	5.0**resolution* + 3.5**storage* - 0.05**price*	$k = 1$
q_2	2.5**resolution* + 7.0**storage* - 0.08**price*	$k = 1$
$\cdots$	$\cdots$	$\cdots$

Figure 1: Example of improvement strategy for cameras

rank becomes higher than that of p_2 for both queries q_1 and q_2.

For ease of presentation, we will simply use $p'_i = p_i + s$ to denote the improved object p'_i that is derived by applying s on p_i. An improvement strategy aims to make a target object appear in more query results. Given an improvement strategy s, we measure its effectiveness in improving object p_i as the *number of top-k queries hit by* $p'_i = p_i + s$, denoted by $H(p'_i)$. A larger $H(p'_i)$ means more effective that s is in improving p_i.

Improving an object requires resources such as time and money. We let the query issuer specify such resource requirements using a *cost function* $Cost_{p_i}(s)$, which computes the cost of applying strategy s to object p_i. There is rich literature on how to model product costs using math functions and interested readers are referred to [19, 4, 2] for details. Here we simply assume the cost functions are provided by the query issuer. Our research is aimed at finding two kinds of improvement strategies:

Definition 2 (Min-Cost Improvement Strategy). *Given an improvement goal that is to hit at least $\tau \in \mathbb{I}$ queries, an improvement strategy s for p_i is a **minimal cost improvement strategy** w.r.t. some cost function $Cost_{p_i}$ if $H(p_i + s) \geq \tau$ and $Cost_{p_i}(s)$ is minimized.*

Definition 3 (Max-Hit Improvement Strategy). *Given a budget $\beta \in \mathbb{R}$, an improvement strategy s for p_i is a **maximal hit improvement strategy** w.r.t. some cost function $Cost_{p_i}$ if $Cost_{p_i}(s) \leq \beta$ and $H(p_i + s)$ is maximized.*

Accordingly, we define two types of *Improvement Queries* (IQs). A *Min-cost IQ* let user query minimal cost improvement strategies for selected objects. Similarly, a *Max-Hit IQ* returns the maximal hit improvement strategies. We will show later in Section 4 that searching for the two types of improvement strategies are NP-Hard even for one target object, and the problem becomes more complex when trying to improving multiple target objects. As such, our goal is to develop highly efficient heuristic algorithms.

3.2 Interpreting Objects as Functions

Our key idea is to interpret each object *as a function* and treat each top-k query *as a function input.* This is different from existing works where queries are considered as utility functions and objects as their input. We use the most common *linear* utility functions[7, 6, 11, 26] as an example to explain our idea.

For linear utility functions, each query $q_i \in Q$ is a d-dimensional vector $q_i = \{q_i^{(1)}, q_i^{(2)}, ..., q_i^{(d)}\}$ that assigns a weight to each attribute of an object and computes the weighted sum. For simplicity, we use the same assumption as existing works that all queries are *normalized*, i.e., $q_i^{(j)} \in [0, 1]$ for any dimension j. In our solution, we treat each object p_i as a linear function f_i, where $p_i^{(j)}$ is the j-th coefficient. It takes a query q as input and computes the ranking score of p_i:

$$f_i(q) = \textstyle\sum_{j=1}^{d} q^{(j)} p_i^{(j)} \qquad (1)$$

Note that the ranking score is the same as the weighted sum. The difference is that a query is now treated as a function parameter while the object attribute values are treated as function coefficients. As such, the set of objects D is interpreted as a set of functions $D = \{f_1, f_2, ..., f_n\}$. When causing no ambiguity, we will use p_i and f_i interchangeably to refer to the same object. To evaluate a top-k query q, we compute $f_1(q), f_2(q), ..., f_n(q)$ and select the k functions with lowest output values.

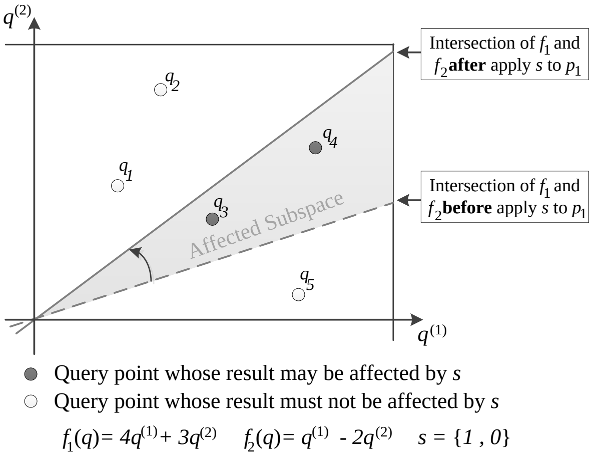
The intersection of two functions f_i and f_j creates a *hyperplane* in the d-dimensional domain space. The intersection partitions the domain into two *subdomains*, namely *above* and *below.* For any input q falls in the above subdomain, we have $f_i(q) \geq f_j(q)$, and for any input q in the below subdomain, we have $f_i(q) < f_j(q)$. The intersections of all functions partition the domain space D into a number of subdomains, and the functions can be strictly sorted in each of these subdomains. That is, if there exists a query point q in a subdomain such that $f_i(q) > f_j(q)$ (or $f_i(q) < f_j(q)$), then for any other query point p in the same subdomain, we have $f_i(p) > f_j(p)$ (or $f_i(p) < f_j(p)$). As a result, the rank of a function f_i remains the same for any two queries q_x and q_y as long as they fall in the same subdomain.

Applying an improvement strategy s to p_i will cause the intersections involving f_i to *tilt* towards some direction determined by s. The boundaries of some subdomains will also move. As showed in Figure 2, it may cause some query points to move to a different subdomain (e.g., move from above to below some intersections). We have two important conclusions.

Fact 1. *An improvement strategy s affects the result of a query q if and only if q is moved to a different subdomain after applying s to p_i. Thus, if no query point is moved to a different subdomain, we have $H(p_i + s) = H(p_i)$.*

Fact 2. *The rank of two functions f_i and f_j must be switched in the ranking result of some query q, if and only if q is moved from above (or below) to below (or above) of the intersection of f_i and f_j.*

The proofs of the two conclusions are straightforward. Due to limited space, we refer readers to [9, 16] for proof details. These facts suggest an efficient way to evaluate a given improvement strategy s. First, we apply s to p_i and


$q^{(2)}$
Intersection of f_1 and f_2 **after** apply s to p_1
q_2
q_4
q_1
Intersection of f_1 and f_2 **before** apply s to p_1
q_3
Affected Subspace
q_5
$q^{(1)}$
● Query point whose result may be affected by s
○ Query point whose result must not be affected by s
$f_1(q) = 4q^{(1)} + 3q^{(2)}$ $f_2(q) = q^{(1)} - 2q^{(2)}$ $s = \{1, 0\}$

Query	Ranking results	
	Before applying s	After applying s
q_1, q_2	$[f_1, f_2]$	$[f_1, f_2]$
q_3, q_4	$[f_1, f_2]$	$[f_2, f_1]$
q_5	$[f_2, f_1]$	$[f_2, f_1]$

Figure 2: An improvement strategy affects subdomain boundaries and query results

find all the query points that are moved to a different subdomain. Then, for each query point found, check if p_i appears in its result and update $H(p_i + s)$ accordingly. The challenge now is, how to efficiently determine (without traverse all query points or subdomains) which query points are moved to which subdomains before and after applying an improvement strategy, and then compute their results. We discuss this approach in detail in the next section.

4. PROPOSED SOLUTION

We first introduce an *Efficient Strategy Evaluation* (ESE), which group query points by subdomains and index them using multidimensional data structures such as R-tree [10] or X-tree [3]. We will then discuss how to use ESE as a building block for efficiently processing of IQs. Here we consider only one target object with linear utility functions. Nevertheless, our techniques allow users to select multiple objects as targets, use different cost functions for each object, and query improvement strategies with non-linear utility functions, which we will discuss later in Section 5.

4.1 Efficient Strategy Evaluation (ESE)

Given an improvement strategy s for p_i, we need to compute its effectiveness in improving p_i, i.e., counting the number of top-k queries that include $p'_i = p_i + s$ in their result. For this purpose, existing solutions such as *Reverse top-k Threshold Algorithm* (RTA) [21] can be used. These schemes, however, support only linear utility functions. In particular, they are less efficient when a less number of queries include the object in their result. When $H(p_i + s)$ increases, their performance will drop significantly. Here we present an approach that works better for our purpose.

Given the intersection of two functions f_i and f_l:

$$\textstyle\sum_{j=1}^{d} q^{(j)}(p_i^{(j)} - p_l^{(j)}) = 0 \tag{2}$$

Equation 3 represents the new intersection hyperplane after some improvement strategy s is applied to p_i.

$$\textstyle\sum_{j=1}^{d} q^{(j)}(p_i^{(j)} + s_j - p_l^{(j)}) = 0 \tag{3}$$

The area bounded between the old and new intersection hyperplanes represented by Equation 2 and 3 formulates a subspace (e.g., the shadow area showed in Figure 2) inside the function domain space. We define this subspace as the *affected subspace* of s. It contains all the query points whose result are affected by applying s to p_i. To efficiently retrieve and evaluate such queries, we group all queries by their subdomains and index them with an R-tree.

Algorithm 1 *FindSubdomains*(I, Q)

```
$d \leftarrow newSubdomain()$
$Subdomains.add(d)$
for all $q \in Q$ do
    $q.subdomain \leftarrow d$
end for
for all $I_i \in I$ do
    for all Subdomain $d \in Subdomains$ such that $d$ overlaps $I_i$ do
        $d_{above} \leftarrow newSubdomain()$
        $d_{above}.boundaries.add(I_i, above)$
        $d_{below} \leftarrow newSubdomain()$
        $d_{below}.boundaries.add(I_i, below)$
        for all $q$ falls in $d$ do
            if $q$ falls above $I_i$ then
                $q.subdomain \leftarrow d_{above}$
            else
                $q.subdomain \leftarrow d_{below}$
            end if
        end for
        if $d_{above}$ contains query then
            $Subdomains.add(d_{above})$
        end if
        if $d_{below}$ contains query then
            $Subdomains.add(d_{below})$
        end if
    end for
end for
Return $Subdomains$
```

Group query points by subdomain: Subdomains are partitioned using intersection hyperplanes of functions in D. Thus we need first to find the intersections created by the functions. This can be efficiently done using intersection discovery algorithms such as the plane sweeping algorithm [15]. We then partition the function domain into subdomains gradually, by considering function intersections one at a time.

Let $I = \{I_1, I_2, ..., I_m\}$ be the set of all function intersections. An intersection hyperplane I_i partitions the domain space into two subdomains: subdomain *above* and subdomain *below* the intersection. As such, it also partitions the query points Q into two groups, above and below. Note that queries fall on the intersection hyperplane can be treated as above it with no affect on the proposed algorithm. Whether a query point q falls above or below I_i is checked as follows. Let I_i be the intersection of some functions f_a and f_b. A query q falls above I_i if and only if $f_a(q) - f_b(q) \leq 0$. Otherwise q is below I_i. These two groups of queries can then be further partitioned by considering another intersection. We repeat this *binary space partitioning* process until no group can be further partitioned. At the end, for each query, we add an attribute *Subdomain* that contains a unique subdomain ID, recording the subdomain that contains the query point. If all query points in a sub-tree have the same *Subdomain* value, then we can mark this on the root-node of the sub-tree, instead of storing the same information for each query point. Note that we can also find which intersection serves as a boundary of a subdomain during this process. Finally, to save space, all the subdomains that contain no query point are simply discarded. A more formal description of this process is given in Algorithm 1.

Once the index is in place, computing $H(p_i+s)$ is straightforward. We only need to evaluate (or re-evaluate, if it is already evaluated) all queries falling in the affected subspaces. To check whether a query point q falls in the affected subspace, it is not necessary to solve the system of Equation 2 and 3. It is determined by two boundary conditions:

$$\textstyle\sum_{j=1}^{d} q^{(j)}(p_i^{(j)} - p_l^{(j)}) \geq 0 \qquad (4)$$

$$\textstyle\sum_{j=1}^{d} q^{(j)}(p_i^{(j)} + s_j - p_l^{(j)}) < 0 \qquad (5)$$

which is equivalent to a range query over the R-tree index, where the query range is the affected subspace (ruled by the boundaries of the function domain, if any). However, evaluating queries in the affected subspace may still be expensive if the affected subspace is large. Here we propose two methods to avoid complete re-evaluation of any query.

First, by Fact 2, if q falls in the affected subspace after s is applied, the new ranking result of q can be generated by simply switching the rank of f_i and f_l in the original ranking result. If q is not in the affected subspace, its result must remain the same. Additionally, if f_l was not in the top-k result of q, it indicates that after applying the improvement strategy, f_i cannot be in the top-k of the q because it only switches order with f_l. As such, we can rapidly eliminate unaffected queries.

Second, all query points fall in the same subdomain share exactly the same ranking result. Thus at most one query needs to be evaluated per subdomain. Recall that we have already grouped query points by their subdomains in the indexing step, and marked for each query which subdomain contains it. Let $TP(p_i) \subseteq Q$ denote the set of queries hit by p_i. The pseudocode of this ESE approach is given in Algorithm 2.

We first find all the affected subspace(s) for the given strategy s. This is done by checking all function intersections involving f_i among the intersections found in the indexing stage. For each query point that falls in an affected subspace of s, we check its query result. If the query has not been evaluated yet, then evaluate it and cache the result for future use (note that at most one query result needs to be cached per subdomain). Otherwise, use the aforementioned function-switching method to rapidly generate its result. For each subdomain, only one query needs to be evaluated, and the result can be shared for all other queries. In ESE, each top-k query needs to be evaluated for at most once, and the result of a large proportion of queries can be generated by re-using the result of their nearby queries, given that they fall in the same subdomain.

4.2 Improvement Strategy Searching

4.2.1 Min-Cost Improvement Strategy

Algorithm 2 *EfficientStrategyEvaluation*(p_i, s)

```
$H(p_i + s) \leftarrow |TP(p_i)|$
for all $f_l \in D$ intersects $f_i$ and $f_l \neq f_i$ do
  Find the affected subspace
  for all $q$ falls in the affected subspace do
    if $q$ is not evaluated then
      evaluate $q$
    end if
    Switch the rank of $f_i$ and $f_l$;
    for all $q_j$ falls in the same subdomain as $q$ do
      if $q_j \notin TP(p_i)$ and $q_j \in TP(p_i + s)$ then
        $H(p_i + v) + +$;
      else if $q_j \in TP(p_i)$ and $q_j \notin TP(p_i + s)$ then
        $H(p_i + v) - -$;
      end if
    end for
  end for
end for
Return $H(p_i + s)$
```

Let p_i be the object to be improved. Given an improvement strategy s, we have the improved object $p_i' = p_i + s$. We use $p_{j,k}$ to denote the k-th ranked object of query q_j. In order for p_i' to be in the result of q_j, the following condition must hold:

$$f_i'(q_j) < f_{j,k}(q_j) \qquad (6)$$

That is, the ranking score of p_i' must be less than that of $q_{j,k}$. Here $f_{j,k}$ is $p_{j,k}$'s corresponding function and f_i' that of p_i'. We have variable $x_j = 1$ if p_i' appears in the result of q_j and $x_j = 0$ otherwise. For the min-cost improvement strategy, the goal is to minimize the cost under the condition that p_i' can hit at least τ queries. This problem can be formulated as a constrained optimization problem:

$$\text{minimize} \quad Cost_{p_i}(s) \qquad (7)$$

$$\text{subject to} \quad \sum_{j=1}^{m} x_j \geq \tau \qquad (8)$$

$$f_i'(q_j) < f_{j,k}(q_j) + (1 - x_j)C \quad \forall j \in [1, m] \quad (9)$$

$$x_j \in \{0, 1\} \quad \forall j \in [1, m] \quad (10)$$

where C denotes a very large number that exceeds the highest score of all objects. Constraint 8 guarantees that the improved object hits at least τ queries, while Constraint 9 ensures that Equation 6 is satisfied for each hit query. Note that the improvement strategy must also be *Valid*. That is, all attribute values of the improved object must not exceed the allowed range. For simplicity, here we assume p_i is defined on $\mathbb{R}^d$, thus the trivial condition $p_i + s \in \mathbb{R}^d$ is omitted in the above formulation. Nevertheless, in the case where this certain limitation on the value of the i-th attribute, additional constraints on s_i can be added to reflect such requirements for valid improvement strategies. For example, if the user does not allow value of the i-th attribute of the target object to be adjusted at all, we can simply add a constraint $s_i = 0$.

The formulated problem is an *integer linear programming* problem [23], which has been studied extensively and no efficient algorithm is known. The problem of searching for the min-cost improvement strategy actually is *NP-hard.* We prove it with a reduction from the Minimal Set Cover problem, which is known to be NP-hard.

Definition 4 (Minimal Set Cover). *Given a set $U = \{u_1, u_2, ..., u_n\}$ and $S = \{S_1, S_2, ..., S_m\}$ where $S_i \subseteq U$. Find the minimal number of subsets in S whose union is U.*

Reduction from Minimal Set Cover to Min-cost Improvement Strategy: An instance of minimal set cover problem can be converted to an instance of the min-cost improvement strategy problem as follows: Create a top-1 query q_i for each element $u_i \in U$ with utility function:

$$u_i(p) = w_{i1} * p^{(1)} + w_{i2} * p^{(1)} + ... + w_{im} * p^{(m)} \quad (11)$$

and set weight w_{ij} to 1 if $u_i \in S_j$, and $w_{ij} = 0$ if otherwise. Suppose the objects are ranked by their utility scores in non-increasing order. Create two m-dimensional objects p_0 and p_1, such that all attributes of p_0 are set to 0 and all attributes of p_1 are set to $1/(m+1)$. Therefore $H(p_0) = 0$ and $H(p_1) = n$. The goal is to improve p_0 such that $H(p_0) = \tau = n$. We impose a simple linear cost function:

$$Cost_{p_0}(s) = s_1 + s_2 + ... + s_m \quad (12)$$

such that the cost of adjusting any attribute of p_1 is equally expensive. Additionally, each attribute of p_0 is discrete and can only be 0 or 1. Note that covering an element $u_i \in U$ is equivalent to hitting query q_i with p_0. In order to do so, an improvement strategy must adjust at least one attribute $p^{(j)}$ of p from 0 to 1 where $w_{ij} = 1$, which indicates that subset S_j should be selected to cover u_i. The total improvement cost is equal to the number of selected subsets. As such, a min-cost improvement strategy for the converted instance can be translated into a minimal set cover for the original instance.□

We now propose a heuristic algorithm (Algorithm 3) which leverages the proposed ESE algorithm to search for the sub-optimal strategy. The algorithm consists of multiple iterations. In each iteration, it first computes for each query $q_j \in Q$, a strategy s_j such that $p'_i = p_i + s_j$ can hit it with the minimal cost. This step generates a set of S of *candidate improvement strategies*. Then we apply to p_i the strategy $s \in S$ with the *minimal cost per hit query* $Cost_{p'_i}(s)/H(p'_i + s)$. Repeat this process until p'_i hits at least τ queries. In each iteration, we call the ESE algorithm as a subroutine to compute $H(p'_i + s_j)$.

Algorithm 3 *MinCostIQ*$(p_i, \tau, Cost_{p_i})$

```
$p'_i \leftarrow p_i$
while $H(p'_i) < \tau$ do
    $S \leftarrow \emptyset$
    for each query $q_j \in Q$ and $\notin TP(p'_i)$ do
        $s_j \leftarrow \arg\min Cost_{p'_i}(s)$ such that $q_j \in TP(p'_i + s)$
        Compute $H(p'_i + s_j)$
        $S.add(s_j)$
    end for
    Find $s \in S$ with minimal $Cost_{p'_i}(s)/H(p'_i + s)$
    if $H(p'_i + s) \leq \tau$ then
        $p'_i = p'_i + s$
    else
        Return $s \in S$ with minimal $Cost_{p'_i}(s)$ and $H(p'_i + s) \geq \tau$
    end if
end while
Return $s = p'_i - p_i$
```

Note that the algorithm requires to find the minimal cost strategy s_j that hits a query q_j. It formulates a single-constraint optimization problem:

$$\text{minimize} \quad Cost_{p_i}(s) \quad (13)$$

$$\text{subject to } f'_i(q_j) < f_{j,k}(q_j) \quad (14)$$

which can be efficiently solved using standard math tools like [12].

The proposed algorithm can be considered a greedy one, since it always selects the improvement strategy with maximal efficiency-cost ratio at each step. The rational behind the algorithm is based on the following observation: The *average cost* per hit query is minimized in a min-cost improvement strategy, comparing with any other improvement strategies that hit the same number of queries. The proposed algorithm tries to minimize the average cost per hit query at each iteration. This greedy method reduces size of the searching space to $O(m)$ per iteration, and the number of iterations is bounded by τ. In comparison, exhaustive search takes at least $O(2^m)$ steps.

Similar to other greedy algorithms, our algorithm may terminate with a local optimum. Nevertheless, our experiment shows the algorithm is efficient enough to answer users' IQs interactively (i.e., a user hardly feels waiting time) with a regular desktop computer. Although the cost of the improvement strategy found may be sub-optimal, it greatly outperforms other methods such as simple greedy search (i.e., always try to hit the query with the least cost, repeat until hit enough queries) and random search (i.e., return a randomly generated improvement strategy), which we will discuss later. To sum up, this algorithm offers a good trade-off between improvement cost and feasibility.

Processing Min-Cost IQs: To issue a min-cost IQ, the query issuer first defines a cost function $Cost_{p_i}$ for the selected target p_i and specifies a desired τ. The system then uses Algorithm 3 to find the improvement strategy that satisfies the desired number of hits. For query issuers who indeed want the optimal strategy, we also provide them with the option of exhaustively strategy searching, which uses mathematical optimization tools (e.g., [12]) to solve the above optimization problem. However, due to the intractability of the problem, this algorithm is only feasible for very small datasets.

4.2.2 Max-Hit Improvement Strategy

Recall that the goal of maximal hit improvement strategy is to maximize the number of queries hit by the improved object with the constraint that the total cost does not exceed a given budget β. Similarly, we formulate the following optimization problem.

$$\text{maximize } H(p_i + s) \quad (15)$$

$$\text{subject to } Cost_{p_i}(s) \leq \beta \quad (16)$$

$$f'_i(q_j) < f_{j,k}(q_j) + (1 - x_j)C \quad \forall j \in [1, m] \quad (17)$$

$$x_j \in \{0, 1\} \quad \forall j \in [1, m] \quad (18)$$

The target function computes the hit number of the improved object, while Constraint 16 corresponds to the limited budget. The meaning of Constraint 17 is the same as the minimal cost improvement strategy problem. It is easy to see that searching for maximal hit improvement strategy is also NP-Hard, because the minimal cost improvement strategy problem reduce to it.

Reduction from Min-Cost Improvement Strategy to

Max-Hit Improvement Strategy: Let $MaxHit$ (p_i, β, $Cost_{p_i}$) be a subroutine that finds the maximal hit improvement strategy. We show how to find the minimal cost improvement strategy for p_i with desired hit τ by calling the subroutine. Let x_{max} be the cost required to hit all top-k queries, which can be treated as a constant. The minimal cost that we are looking for must fall in $[0, x_{max}]$, so we can search for the minimal cost strategy with a binary searching process. We start by setting β to an initial value x such that $x_{max} \geq x \geq 0$, and use the subroutine to find s such that $p_i + s$ hit the maximal number of queries. If $H(p_i + s) \geq \tau$, it means the minimal cost required to hit τ queries is no greater than x. Thus we refine the searching range by setting β to a new value in $[0, x]$ and repeat the process. Similarly, if $H(p_i + s) < \tau$, it indicates the minimal cost required must be larger than x and thus we set β to a new value in $[x, x_{max}]$. Regardless of the initial value, this binary searching process can find the minimal cost improvement strategy within $\log x_{max}$ attempts (i.e., by calling $MaxHit(p_i, \beta)$ for at most $\log x_{max}$ times, which is linear).□

The above proof demonstrates that the two improvement strategies, namely min-cost and max-hit, are closely related to each other. The two types of improvement strategies share a similar characteristic: the cost per hit query is minimized for a max-hit improvement strategy, comparing with any other improvement strategies with the same cost. As such, we modify the greedy searching Algorithm 3 to process max-hit IQs. The algorithm uses a similar searching method which looks for the most cost-efficient improvement strategy in each iteration, and the iterations terminate when all budget is used, or there is not enough budget to cover more queries.

Algorithm 4 *MaxHitIQ*$(p_i, \beta, Cost_{p_i})$

```
$p'_i \leftarrow p_i$
$s* \leftarrow 0$
while $Cost_(p_i)(s*) < \beta$ do
    $S \leftarrow \emptyset$
    for each query $q_j \in Q$ and $\notin TP(p'_i)$ do
        $s_j \leftarrow \arg\min Cost_{p'_i}(s)$ such that $q_j \in TP(p'_i + s)$
        Compute $H(p'_i + s_j)$; $S.add(s_j)$
    end for
    Find $s \in S$ with minimal $Costp'_i(s)/H(p'_i + s)$
    if $Cost_(p_i)(s*) + Cost_(p_i)(s) \leq \beta$ then
        $s* += s$
    else
        for each $s \in S$, sorted by cost do
            if $Cost_(p_i)(s*) + Cost_(p_i)(s) \leq \beta$ then
                $s* += s$
            end if
        end for
        Break
    end if
end while
Return $s*$
```

Processing Max-Hit IQs: A max-hit IQ consists of target object(s), corresponding cost function(s), and a budget β. The improvement strategy that satisfies the budget constraint is then returned to the user by Algorithm 4. For convenience, we will refer to Algorithms 3 and 4 together as the **Efficient-IQ** algorithm. Similarly, we also provide the exhaustive search option in our implementation.

4.3 Data updating

Add/Remove a query: When a query point is added to or removed from Q, the R-tree needs to be updated. Adding or removing an indexed point on R-tree is easy. However, when a new query point is added, we need to find which subdomain contains it. We can use Algorithm 1 but only on the newly added query point to find its subdomain. This is usually not necessary. We observe that, if a new query point q falls closely to a group of other query points which are all in a subdomain d, then it is very likely that q also falls in d. Fortunately, we can quickly check if q falls in d by verifying the above/below relations between q and the boundary intersections of d as in Algorithm 1. Based on this observation, we propose to use the subdomain(s) of the k-Nearest Neighbour of q as candidate subdomain of q, and use Algorithm 1 only if q is not in any of these candidates.

Add/Remove an object: Adding or removing an object will cause the boundary of subdomains to change. Thus, similar to applying an improvement strategy, some query points may move to a different subdomain. We discuss how to update subdomain of affected queries as follows. When a new object is added, we first find all the newly created intersections and then rerun Algorithm 1 with these intersections to update the queries. Similarly, when an object is removed, we find all existing intersections that involve the object, and then locate all subdomains whose boundaries include one of the involved intersections. Then, if the subdomain is above the intersection, we merge it with the subdomain that is below it, and vice versa. This is to reflect the fact the once the object is removed, this intersection no longer exists, and the two subdomains that were separated by it should be merged as one subdomain. To facilitate this process, we implement a bloom filter to index the subdomains based on their boundaries, allowing us to quickly check if a subdomain uses an intersection as its boundary.

5. EXTENSION

5.1 Improving Multiple Target Objects

So far we have considered improving a single object. In this section, we extend our proposed techniques to enable users to query strategies that improve multiple objects. Here a user wants to select a set of objects $D_t \subseteq D$ as targets, and query the min-cost improvement strategy such that the total number of hits of the targets is no less than certain threshold τ, while the total improving cost is minimized. Each target can be associated to a different cost function, or share the same one. We assume that if one query is hit by two different target objects in D_t, the query is counted only once. We consider two **Combinatorial Object Improvement** problems.

DEFINITION 5. *Given a set of target objects $D_t \subseteq D$ and their corresponding cost functions, the* ***Combinatorial Min-Cost Improvement Strategy*** *for D_t is a set of improvement strategies S_t, where $s_i \in S_t$ is an improvement strategy for $p_i \in D_t$, such that $\sum_{p_i \in D_s} H(p_i + s_i) \geq \tau$ and $\sum_{p_i \in D_s} Cost_{p_i}(s_i)$ is minimized.*

DEFINITION 6. *Given a set of target objects $D_t \subseteq D$ and their corresponding cost functions, the* ***Combinatorial Max-***

Hit Improvement Strategy *for D_t is a set of improvement strategies S_t, where $s_i \in S_t$ is an improvement strategy for $p_i \in D_t$, such that $\sum_{p_i \in D_s} Cost_{p_i}(s_i) \leq \beta$ and $\sum_{p_i \in D_s} H(p_i + s_i)$ is maximized.*

The two problems are both NP-hard, since the single-object improvement strategy problems are their special cases. We can slightly modify the algorithms proposed in Section 4.2 to handle the combinatorial improvement strategy searching problems. To search for the combinatorial minimal cost improvement strategy, we can modify Algorithm 3 as follows: First finds the min-cost improvement strategies that can hit each query, and uses them as candidates. The algorithm then selects the candidate strategy with minimal cost per hit query. This process is repeated until at least the desired number of queries are hit. A more formal description is given as follows:

- Step 1: For each query q and each target object p_i, find the minimal-cost improvement strategy that makes p_i hits q. All such improvement strategies are used as candidates.
- Step 2: Find and apply the candidate strategy s with minimal cost per hit query. If the total number of hit queries after applying the strategy is larger than τ, then instead of s, we should apply the candidate strategy that hits at least τ queries with minimal cost. This is to avoid over-achieving the desired number of hits, and thus increase the total cost.
- Step 3: If the number of query hit by the improved objects is less than τ, repeat step 1 and 2.

Similarly, for max-hit IQ, we modify Algorithm 4 to make it applicable for multiple target objects.

- Step 1: For each query q and each target object p, find the minimal-cost improvement strategy that makes p_i hits q. All such improvement strategies are used as candidates.
- Step 2: Filter out the candidate strategies whose cost exceeds the remaining budget. If the candidate set is not empty, then select the candidate strategy with minimal cost per hit query, and apply it to the corresponding object. Update the remaining budget accordingly. If the candidate set is empty, then terminate.
- Step 3: If there is still available budget, repeat step 1 and 2.

5.2 Complex Utility Functions

We now discuss how to handle the case when the utility functions used in top-k queries are non-linear. Regardless of its complexity, a utility function $f(p_i)$ can always be seen as a function $f_{p_i}(q)$ for object p_i, in which the attribute values of p_i are treated as constants of the function, while the variable q consists of the other parameters of the top-k query (e.g., attribute weights as in linear utility functions). We explain the idea with a complex utility function example, applied on a Car dataset with three attributes (Table 1), where w_1 and w_2 are user-specified weights.

$$u(Car \quad c) = \sqrt{w_1 * c.Price} + w_2 \frac{c.Capacity}{c.MPG} \tag{19}$$

As showed in the table, each car object can be seen as a non-linear function, by treating its *Price*, *MPG* (Mileage Per Gallon gas), and *Capacity* as constants. The function has input variables (w_1, w_2). The intersection of non-linear functions can take a more complex form. Generally, the intersection of two d-variable functions formulates a *surface* in the d-dimensional domain space. Nevertheless, our observation that these functions are sortable in subdomains partitioned by their intersection is still valid. Thus the proposed Efficient-IQ algorithm works as well over complex functions. Our concern is, however, for certain complex functions, the number of subdomains partitioned by intersections can be very large [1], which may result in a high indexing cost.

Table 1: *Car* dataset and the corresponding functions

ID	Price	MPG	Capacity	$u(w_1, w_2)$
1	15000	30	4	$\sqrt{15000w_1} + w_2\frac{4}{30}$
2	20000	28	6	$\sqrt{20000w_1} + w_2\frac{6}{28}$
3	8000	35	2	$\sqrt{8000w_1} + w_2\frac{2}{35}$

To mitigate this problem, we propose to *convert non-linear functions into linear functions* through variable substitution, i.e., replacing complex components of an equation with one variable to simplify the equation. After converting non-linear functions into linear ones, we can then apply the same techniques introduced in Section 4 for efficient processing of IQs. Consider an example of top-k queries with polynomial utility function, applied on a 4-dimensional dataset D:

$$u(p) = w_1(p^{(1)})^3 + w_2(p^{(2)} * p^{(3)}) + w_3(p^{(4)})^2 \tag{20}$$

which contains three high degree terms. It can be converted into an equivalent linear function:

$$u^*(p) = w_1 p^{(5)} + w_2 p^{(6)} + w_3 p^{(7)} \tag{21}$$

where $p^{(5)} = (p^{(1)})^3$, $p^{(6)} = p^{(2)} * p^{(3)}$, and $p^{(7)} = (p^{(4)})^2$ are used to substitute $p^{(1)}$-$p^{(4)}$. As such, each object becomes 7-dimensional. Nevertheless, in this example, attributes 1 4 are no longer used in the converted utility function, thus the dataset can be treated as 3-dimension. The value of each augmented attributed is computed using the original attribute values of the object, thus they do not need to be computed and stored in advance. Instead, we simple store the conversion process as math formulas, and compute their values on the fly to avoid storage redundancy.

Variable substitution can be used to convert other forms of complex functions into linear ones as well. Consider function:

$$u(p) = \sqrt{(w_1 - p^{(1)})^2 + (w_2 - p^{(2)})^2} \tag{22}$$

which computes the Euclidean distance between a data point and a given location $\{w_1, w_2\}$. We can make the following conversion:

$$u^*(p) = (w_1 - p^{(1)})^2 + (w_2 - p^{(2)})^2 \tag{23}$$

$$u^*(p) = (w_1^2 + w_2^2) - 2w_1 p^{(1)} - 2w_2 p^{(2)} \tag{24}$$

$$+ p^{(3)} + p^{(4)} \tag{25}$$

where $p^{(3)} = (p^{(1)})^2$ and $p^{(4)} = p^{(2)})^2$ are the two augmented attributes. Note that $u^*(p) = u(p)^2$. Since distance is always positive, the ranking result of the converted function

[1] For linear functions, the number of such subdomains is bounded by $O(n^d)$ where n is the number of objects and d the number of variables [17]. While for some high-degree functions, the number can be $O(2^n)$.

remains the same.

5.3 Heterogeneous Utility Functions

Since IQ allows users to apply complex utility functions, it is possible that each user defines a utility function with a completely different form. For example, to query the Car dataset (Table 1), some users may express their preference as a different utility function:

$$v(Car\quad c) = \frac{c.MPG}{w_1 * c.Price} + w_2(c.Capacity)^2 \qquad (26)$$

In this case, we cannot simply use the value of (w_1, w_2) to differentiate different top-k queries. Because even for the same (w_1, w_2), the two functions 19 and 26 may compute different values, as they represent two evaluation methods over the same dataset. The default way to handle heterogeneous utility function is to add another column $v(w_1, w_2)$ to the Car dataset, and use function outputs in this column to sort the objects when considering the top-k queries with $v(Car\quad c)$. However, this will significantly increase the indexing cost, because we need to find subdomains for two different sets of functions, each has the same size of the object set.

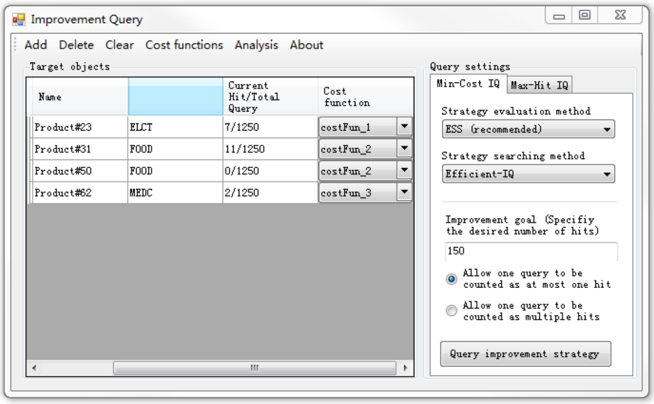
To address this problem, we propose constructing a "generic" function in such a way that all the user-defined utility functions are special cases of this one function. Let's continue with the Car dataset example. Construct the following generic function for functions 19 and 26 by adding them up:

$$\begin{aligned} G(Car\quad c) =& u(Car\quad c) + v(Car\quad c) && (27)\\ =& \sqrt{w_1 * c.Price} + w_2\frac{c.Capacity}{c.MPG} && (28)\\ +& \frac{c.MPG}{w_3 * c.Price} + w_4(c.Capacity)^2 && (29) \end{aligned}$$

Now we can differentiate two queries by the value of (w_1, w_2, w_3, w_4) as in the linear case. Our solution works because if a query uses function 19 as utility function, it must set w_3, w_4 to 0. While for queries with function 26, w_1, w_2 is 0. As such, we unify the domain of the two functions into one domain space, and are able to interpret each object as only one function.

6. IMPLEMENTATION AND EVALUATION

6.1 System Implementation



Name		Current Hit/Total Query	Cost function
Product#23	ELCT	7/1250	costFun_1
Product#31	FOOD	11/1250	costFun_2
Product#50	FOOD	0/1250	costFun_2
Product#62	MEDC	2/1250	costFun_3

Strategy evaluation method: ESS (recommended)

Strategy searching method: Efficient-IQ

Improvement goal (Specifiy the desired number of hits): 150

● Allow one query to be counted as at most one hit

○ Allow one query to be counted as multiple hits

Figure 3: Graphic User Interface for Improvement Query

We have implemented the proposed techniques as an analytic tool and integrated it with the Database Management System (DBMS). The tool allows users to issue IQs in an interactive way via a Graphic User Interface (GUI) showed in Figure 3. Users can select target objects manually from the object dataset or via an SQL select statement. For the target objects, users specify which attributes can be adjusted and in what range, and also the cost function to be used for each object. Our system is implemented using C++ and C# on a Windows server with Intel Xeon 64-bit 8-core CPU running on 2.93GHz and 32GB RAM. An R-tree is used to index the queries. For comparison purpose, we implement four IQ processing schemes in our experiments.

- **Efficient-IQ:** This is the proposed heuristic algorithm, which uses the ESE algorithm for improvement strategy evaluation.
- **RTA-IQ:** This implementation uses the RTA algorithm, designed for reversed top-k query, to evaluate improvement strategies in each iteration, instead of the proposed ESE algorithm. Note that RTA supports only linear utility functions.
- **Greedy:** This implementation uses simple greedy algorithm. It always finds the query point that can be hit by any target object with the minimal cost, then repeats the process until the desired number of queries are hit (for Min-Cost IQs), or there is no budget left (for Max-Hit IQs).
- **Random:** This scheme randomly generates improvement strategies until it finds an improvement strategy that satisfies the improvement goal (i.e., hits the desired number of queries, or total cost less than the budget), and returns it as the answer to user's IQ.

6.2 Data Preparation

We test our system over four types of object datasets, namely Independent (IN), Correlated (CO), Anti-correlated (AC), and Real-world. IN, CO, and AC are synthetic datasets generated with the method described in [5]. Specifically, in IN, all attributes of an object are generated independently with a uniform distribution, while in CO and AC, attribute values of the an object is correlated or anti-correlated, respectively. Each generated object has 10 numerical attributes in range $[0, 1]$. We use two real-world datasets: VEHICLE and HOUSE. VEHICLE [1] contains 37051 vehicle models with attributes including year, weight, horse power, mileage per gallon (MPG), and annual cost. HOUSE is extracted from [18], including 100,000 records with four attributes house value, household income, number of person, and monthly mortgage payment. We normalize attributes of the real-world datasets to $[0, 1]$.

We generate two sets of top-k queries, namely UN and CL. Both sets of queries use polynomial utility functions, while the distribution of function coefficients (weights) are uniform and independent in UN but clustered in CL. Details of how to generate such queries are given in [21]. The degree of each term in the function is randomly chosen from $[1, 5]$ and the top-k value is randomly selected from $[1, 50]$. The default experiment setting is given in table 2.

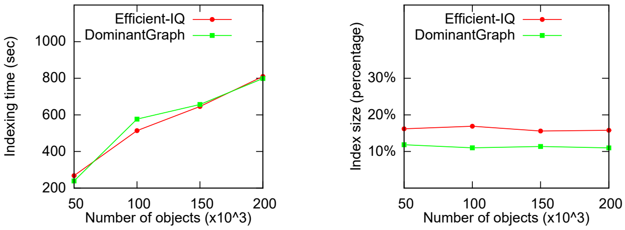
6.3 Experiment Results

Table 2: Experiment Setting

Parameter	Default	Range
$\|D\|$	100,000	50,000 - 200,000
$\|Q\|$	10,000	5,000 - 15,000
τ	250	100 - 500
β	50	10 - 100
Dimensionality	3	1 - 5

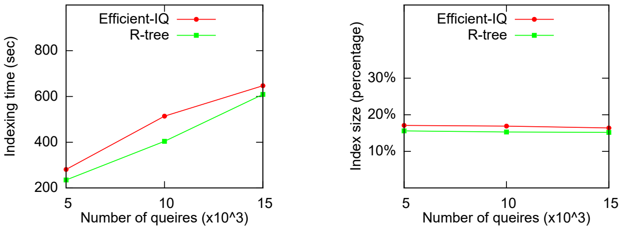
6.3.1 *Data Indexing*

We first evaluate the indexing cost of the proposed techniques, which involve the cost of building an R-tree over the query points and grouping them by subdomains. To better understand the scale of this cost, we compare indexing structure size (showed as percentage to the original dataset) and the total indexing time of the proposed technique (**Efficient-IQ**) with two benchmarks: 1) the cost of building only an R-tree on the query points (**R-tree**), and 2) the cost of building a Dominant Graph (**DominantGraph**) [26] for the objects, which is the state-of-the-art indexing technique for top-k query with linear utility functions.



(a) Indexing time (b) Index size

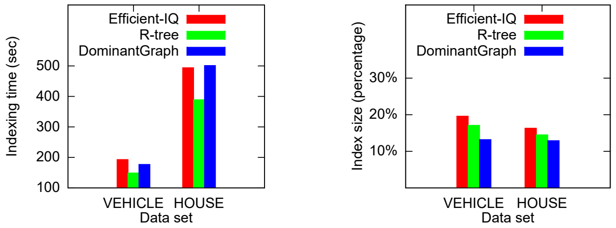
Figure 4: Scalability to the object set size



(a) Indexing time (b) Index size

Figure 5: Scalability to the query set size

We adjust the number of objects and report the corresponding indexing time and size of the proposed technique and DominantGraph (Figure 4). In order for Dominant Graph to work, we use only linear utility functions for top-k queries. For each test point, we generate 100 different utility functions and report the average indexing costs. We observe that the indexing cost on different types of synthetic data is almost the same, thus we report the average cost over all the types of datasets to save space. The dimension (i.e., number of variables) of the utility functions is uniformly picked in [1, 5]. The indexing time of DominantGraph is similar to our technique in general while Efficient-IQ incurs slightly higher storage overhead (less than 5% of the data size). However, our technique is unique in being able to support efficient processing of IQ.



(a) Indexing time (b) Index size

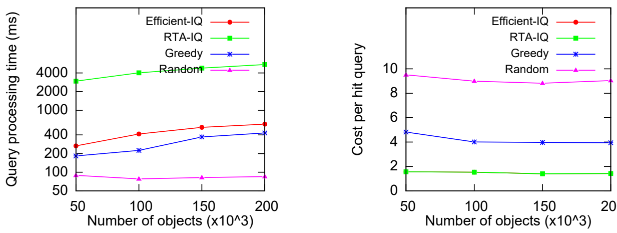
Figure 6: Indexing cost of real-world datasets

We then adjust the number of queries and compare the proposed technique with building only an R-tree (Figure 5). This time we allow non-linear utility functions. For the same set of queries, the proposed Efficient-IQ requires about 20% - 25% more indexing time comparing with building only an R-tree. The extra time is used to find subdomains for each query point, in order to facilitate the ESE algorithm. The final index size, nevertheless, is only about 10% larger than an R-tree. This is because many adjacent query points fall in the same subdomain and thus we do not need to store the subdomain information for each of them. In general, the propose technique shows good scalability, in terms of indexing cost, with respect to both the number of objects and queries. Experiment over real-world datasets is consistent with that on synthetic data.

6.3.2 *IQ Processing*

For query processing, we are interested in two metrics: 1) Average query processing time, and 2) Quality of the improvement strategy returned to the user. For Min-Cost IQ, the quality of an improvement query can be measured by its total cost. While for Max-Hit IQ, it's the total number of query hit by the improved objects. We use an unified quality measurement for both types of queries, i.e., the average cost per hit query of an improvement strategy, the lower the better. If multiple target objects hit the same query, we count them as only one hit. Our experiment shows that, even for the smallest dataset, exhaustive search takes more than 4 hours to process a query in average. Thus we compare only the 4 aforementioned schemes. For RTA-IQ to work, we limit the type of utility functions to linear with attribute weights normalized to 1. We use the following cost function for all objects:

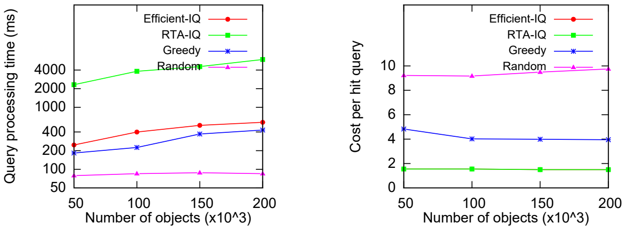
$$Cost(s) = \sqrt{\sum_{i=1}^{d} s_i^2} \tag{30}$$



(a) Query processing time (b) Cost per hit query

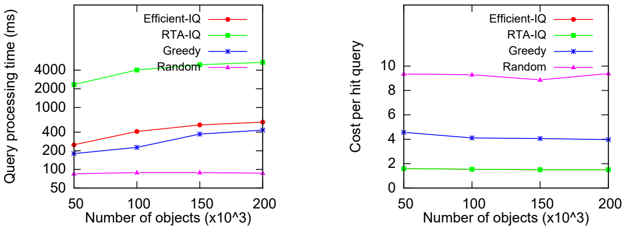
Figure 7: Query processing on the IN object dataset

We evaluate the scalability of the proposed techniques with regard to the size of D and Q respectively. The results on different data sets are showed in Figure 7-13. For



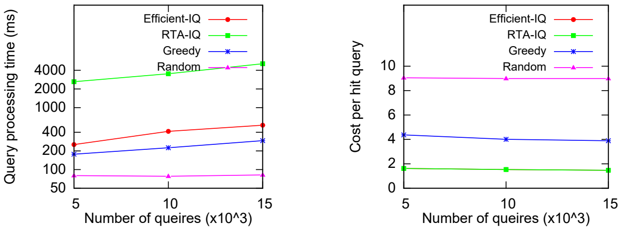
(a) Query processing time (b) Cost per hit query

Figure 8: Query processing on the CO object dataset



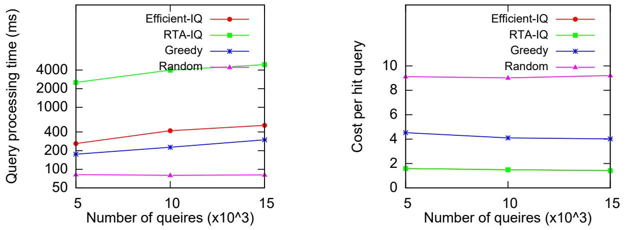
(a) Query processing time (b) Cost per hit query

Figure 9: Query processing on the AC object dataset



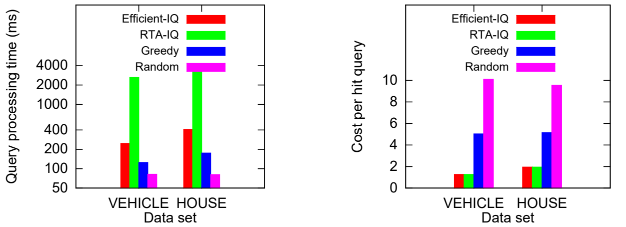
(a) Query processing time (b) Cost per hit query

Figure 10: Query processing on the UN query dataset



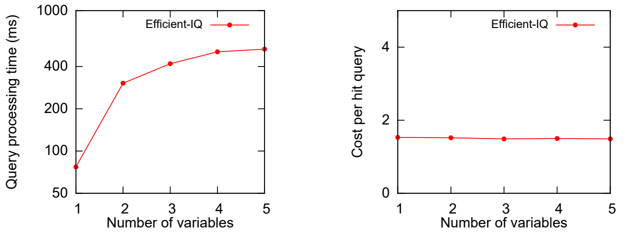
(a) Query processing time (b) Cost per hit query

Figure 11: Query processing on the CL query dataset



(a) Query processing time (b) Cost per hit query

Figure 12: Query processing on the real-world datasets



(a) Query processing time (b) Cost per hit query

Figure 13: Scalability to the number of variables in functions

each test point, we issue 100 Min-Cost IQs and 100 Max-Hit IQs, and report the average performance of the compared schemes. The parameters of these IQs are randomly and uniformly selected from the ranges given in Table 2. For each real-world dataset, we use a randomly generate query set that is one third of its size.

It is not surprising that Random is the fastest scheme in processing IQs, but it also yields the worst improvement strategy quality. The simple greedy algorithm has better strategy quality than Random, but is still very poor when compared with the proposed techniques. The Efficient-IQ achieves both good running time and high strategy quality. It outperforms RTA-IQ significantly in querying processing time, while achieving the best improvement strategy quality. (Note that RTA-IQ uses the same strategy-searching approach as Efficient-IQ, thus the quality of the strategies found by the two schemes is the same). The result shows that the good performance of the proposed technique is due to the combination of an efficient strategy searching method and a fast evaluation algorithm used in each searching iteration.

Finally, we evaluate the scalability of the proposed technique with respect to dimensionality of the functions (i.e., the number of variables in the interpreted functions). Since RTA only works on linear function, in this experiment we plot only the result of Efficient-IQ. The result (Figure) shows as the number of variables increases, the query processing time increases too, but in a sub-linear way. That means the query processing time becomes less sensitive to dimensionality as it increases, which is a desired feature.

7. CONCLUSION

We live in a society that is competitive in nature. Daily we face the challenges of improving something to make it more competitive against its peers. In this paper, we consider the problem of finding improvement strategies. We propose a new type of query called *Improvement Query* (IQ) that has two variants. A Min-Cost IQ retrieves the improvement strategy with minimal cost for some target object to hit a desired number of top-k queries, and a Max-Hit IQ tries to find an improvement strategy that maximize the number of hit queries with a given budget. Here the cost of an improvement strategy is modeled by a user-defined cost function. We show that finding the exact answers to both queries are NP-Hard and propose a suite of heuristic solutions. Our key idea is to interpret each object as a function and treat each top-k query as as its input. As such, the set of functions can be strictly sorted by their output in each subdomain partitioned by their intersections. The geometrical relations among then function intersections can then be leveraged for efficient processing of IQs. We implement the proposed techniques as an analytic tool and integrated it with the DBMS. In our extensive evaluation, it demonstrates excellent performance.

8. REFERENCES

[1] Fueleconomy.gov vehicle data. http://www.fueleconomy.gov/feg/ws/index.shtml. Updated: Tuesday April 12 2016.

[2] J. Anderson. Determining manufacturing costs. *CEP*, pages 27–31, 2009.

[3] S. Berchtold, D. Keim, and H. Kriegel. An index structure for high-dimensional data. *Readings in multimedia computing and networking*, page 451, 2001.

[4] B. R. Binger et al. *Microeconomics with calculus.* Number 338.501 B613 1998. Addison-Wesley, 1998.

[5] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 421–430. IEEE, 2001.

[6] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: indexing for linear optimization queries. In *ACM SIGMOD Record*, volume 29, pages 391–402. ACM, 2000.

[7] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *VLDB*, volume 99, pages 397–410, 1999.

[8] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *Proceedings of the 32nd international conference on Very large data bases*, pages 451–462. VLDB Endowment, 2006.

[9] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 2000.

[10] A. Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[11] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. In *ACM SIGMOD Record*, volume 30, pages 259–270. ACM, 2001.

[12] L. G. Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, 1980.

[13] H. Lu and C. S. Jensen. Upgrading uncompetitive products economically. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 977–988. IEEE, 2012.

[14] K. Mouratidis, J. Zhang, and H. Pang. Maximum rank query. *Proceedings of the VLDB Endowment*, 8(12):1554–1565, 2015.

[15] J. Nievergelt and F. P. Preparata. Plane-sweep algorithms for intersecting geometric figures. *Communications of the ACM*, 25(10):739–747, 1982.

[16] F. P. Preparata and M. Shamos. *Computational geometry: an introduction.* Springer Science & Business Media, 2012.

[17] L. Schläfli. *Theorie der vielfachen Kontinuität*, volume 38. Zürcher & Furrer, 1901.

[18] R. G. J. G. Steven Ruggles, Katie Genadek and M. Sobek. *Integrated public use microdata series: Version 6.0 [Machine-readable database].* University of Minnesota, 2015.

[19] J. Viner. *Cost curves and supply curves.* Springer, 1932.

[20] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 365–376. IEEE, 2010.

[21] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Norvag. Monochromatic and bichromatic reverse top-k queries. *Knowledge and Data Engineering, IEEE Transactions on*, 23(8):1215–1229, 2011.

[22] Q. Wan, R. C.-W. Wong, I. F. Ilyas, M. T. Özsu, and Y. Peng. Creating competitive products. *Proceedings of the VLDB Endowment*, 2(1):898–909, 2009.

[23] L. A. Wolsey and G. L. Nemhauser. *Integer and combinatorial optimization.* John Wiley & Sons, 2014.

[24] Z. Zhang, S.-w. Hwang, K. C.-C. Chang, M. Wang, C. A. Lang, and Y.-c. Chang. Boolean+ ranking: querying a database by k-constrained optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 359–370. ACM, 2006.

[25] Z. Zhang, C. Jin, and Q. Kang. Reverse k-ranks query. *Proceedings of the VLDB Endowment*, 7(10):785–796, 2014.

[26] L. Zou and L. Chen. Dominant graph: An efficient indexing structure to answer top-k queries. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 536–545. IEEE, 2008.

Grid-Index Algorithm for Reverse Rank Queries

Yuyang Dong
Department of Computer Science
University of Tsukuba
Ibaraki, Japan
touuyou@gmail.com

Hanxiong Chen
Department of Computer Science
University of Tsukuba
Ibaraki, Japan
chx@cs.tsukuba.ac.jp

Jeffrey Xu Yu
Department of System Engineering and Engineering Management
The Chinese University of Hong Kong, China
yu@se.cuhk.edu.hk

Kazutaka Furuse
Department of Computer Science
University of Tsukuba
Ibaraki, Japan
furuse@cs.tsukuba.ac.jp

Hiroyuki Kitagawa
Department of Computer Science
University of Tsukuba
Ibaraki, Japan
kitagawa@cs.tsukuba.ac.jp

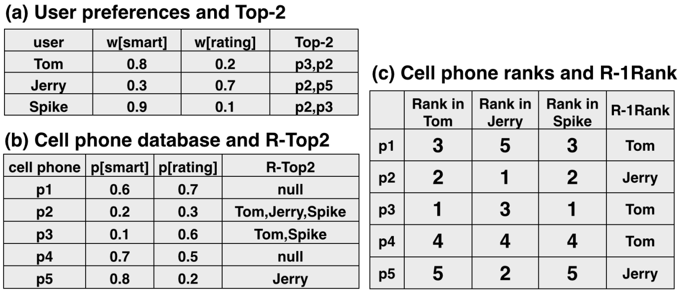
ABSTRACT

In Rank-aware query processing, reverse rank queries have already attracted significant interests. Reverse rank queries can find matching customers for a given product based on individual customers' preference. The results are used in numerous real-life applications, such as market analysis and product placement. Efficient processing of reverse rank queries is challenging because it needs to consider the combination on the given data set of user preferences and the data set of products.

Currently, there are two typical reverse rank queries: Reverse top-k and reverse k-ranks. Both prefer top-ranking products and the most efficient algorithms for them have a common methodology that indexes and prunes the data set using R-trees. This kind of tree-based algorithms suffers the problem that their performance in high-dimensional data declines sharply while high-dimensional data are significant for real-life applications. In this paper, we propose an efficient scan algorithm, named Grid-index algorithm (GIR), for processing reverse rank queries efficiently. GIR algorithm uses an approximate values index to save computations in scanning and only requires a little memory cost. Our theoretical analysis guarantees the efficiency and the experimental results confirm that GIR has superior performance compared to tree-based methods in high-dimensional applications.

CCS Concepts

•**Theory of computation → Database query processing and optimization (theory);**



(a) User preferences and Top-2

user	w[smart]	w[rating]	Top-2
Tom	0.8	0.2	p3,p2
Jerry	0.3	0.7	p2,p5
Spike	0.9	0.1	p2,p3

(b) Cell phone database and R-Top2

cell phone	p[smart]	p[rating]	R-Top2
p1	0.6	0.7	null
p2	0.2	0.3	Tom,Jerry,Spike
p3	0.1	0.6	Tom,Spike
p4	0.7	0.5	null
p5	0.8	0.2	Jerry

(c) Cell phone ranks and R-1Rank

	Rank in Tom	Rank in Jerry	Rank in Spike	R-1Rank
p1	3	5	3	Tom
p2	2	1	2	Jerry
p3	1	3	1	Tom
p4	4	4	4	Tom
p5	5	2	5	Jerry

Figure 1: Example for RTK and RKR queries. (a): the top-2 cell phones appreciated by users. (b): the RT-2 of each phone. (c): the rank list and the $R1$-R of each phone.

Keywords

Reverse Rank Queries; High-dimensional Data Querying;

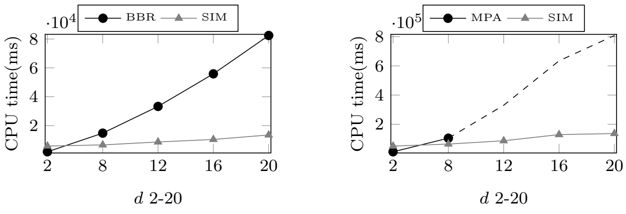
1. INTRODUCTION

Top-k queries retrieve top-k products based on a given user preference. As a user-view model, top-k queries are widely used in many applications as shown in [3, 8]. Assuming that there is a dataset of user preferences, reverse rank queries (RRQ) have been proposed to retrieve the user preference that causes a given product to match the query condition. From the perspective of manufacturers, RRQ are essential to identify customers who may be interested in their products and to estimate the visibility of their fproducts based on different user preferences. Not limited to the field of product (user) recommendations for e-commerce, this concept of user-product can be extended to a wider range of applications, such as business reviewing, dating and job hunting.

Reverse top-k (RTK) [13, 14] and reverse k-ranks (RKR) [22] are two typical RRQ queries. Figure 1 shows an example of RTK and RKR queries. In this example, five different cell phones are scored on how "smart" they are and the "rating". Also, there is a preferences database for three users.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.28



(a) BBR vs SIM.

(b) MPA vs SIM.

Figure 2: Performance of tree-base algorithms (BBR, MPA) and Simple scan on varying d (2-20).

These preferences are based on a series of weights for each attribute. The score of a cell phone based on a user's preference is found by a weighted sum function that computes the inner product of the cell phone attributes vector and the user preferences vector. Without loss of generality, we assume that minimum values are preferable.

From the values in Figure 1, Tom's score for cell phone p_1 is $0.6 \times 0.8 + 0.7 \times 0.2 = 0.62$. All cell phones' scores are calculated in the same way and ranked. If a cell phone is in the top-k of a user's rank list, then the user is in the result of the *RTK* query for that specific cell phone. In Figure 1 (b), the *RT*-2 results for each cell phone are shown. We can see that p_2's *RT*-2 results are Tom, Jerry and Spike, meaning that all users consider p_2 as an element of their top-2 favorites. Notice that p_1 and p_4 have empty *RT*-2 result sets, which means that every user prefers at least two other phones. [22] believed that it was not useful to return an empty answer and proposed *RKR* query, which find the top-k user preferences whose rank for the given product is highest among all users. In Figure 1(c), p_1 is ranked 3rd by Tom, 5th by Jerry, and 3rd by Spike. In other words, Tom (Spike) ranks p_1 higher than other users, so he is in the answer of the *R1-R* of p_1.

1.1 Notations and Problem Definition

Each product p in the data set P is a d-dimensional vector, where each dimension is a numerical non-negative scoring attribute. p can be represented as a point $p = (p[1], ..., p[d])$, where $p[i]$ is an attribute value on ith dimension of p. The data set of preferences, W, is defined in a similar way. w is a user preference vector for products where $w \in W$, and $w[i]$ is the user defined weight value for the attribute on ith dimension, where $w[i] \geq 0$ and $\sum_{i=1}^{d} w[i] = 1$. The score is defined as an inner product of w and p, which is expressed as $f_w(p) = \sum_{i=1}^{d} w[i] \cdot p[i]$. Notations are summarized in Table 1. The definitions of top-k query and of the two reverse rank queries [13, 22] are re-used here.

Definition 1. *(Top-k query): Given a positive integer k, a point set P and a user-defined weighting vector w, the resultant set $TOP_k(w)$ of the top-k query is a set of points such that $TOP_k(w) \subseteq P$, $|TOP_k(w)| = k$ and $\forall p_i, p_j$: $p_i \in TOP_k(w)$, $p_j \in P - TOP_k(w)$. Therefore, it holds that $f_w(p_i) \leq f_w(p_j)$.*

Definition 2. *(*RTK *query): Given a query point q and k, as well as P and W (dataset of points and weighting vectors respectively), a weighting vector $w_i \in W$ belongs to the reverse top-k result set of q, if and only if $\exists p \in TOP_k(w_i)$ such that $f_{w_i}(q) \leq f_{w_i}(p)$.*

Symbol	Description
d	Data dimensionality
P	Data set of products (points)
W	Data set of weighting vectors
q	Query point
$f_w(p)$	The score of p based on w, $f_w(p) = \sum_{i=1}^{d}(w[i] \cdot q[i])$.
$p[i]$	Value of a point $p \in P$ on i'th dimension
$p^{(a)}$	Approximate index vector of a point p
$P^{(A)}$	Approximate index vectors set $\forall p \in P$
n	Number of partitions of value range
$Grid$	Grid-index
$L[f_w(p)]$	Lower bound of score of p on w
$U[f_w(p)]$	Upper bound of score of p on w
$q \prec_w p$	q precedes p based on w

Table 1: Notations and symbols

Definition 3. *(*RKR *query): Given a query point q and k, as well as P and W, reverse k-ranks returns a set S, where $S \subseteq W$ and $|S| = k$, such that $\forall w_i \in S, \forall w_j \in (W - S)$, $rank(w_i, q) \leq rank(w_j, q)$.*

The $rank(w, q)$ is defined as the number of points with a smaller score than q for a given w.

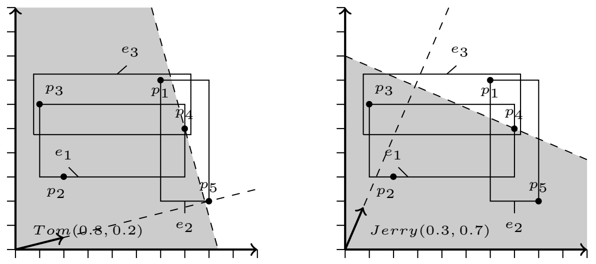
1.2 Motivation and Challenges

To the best of our knowledge, the most efficient algorithm for processing *RTK* is the Branch-and-Bound (BBR) algorithm [17], and the most efficient algorithm for *RKR* is the Marked-Pruning-Approach (MPA) algorithm [22]. Both algorithms use a tree-based methodology, which uses an R-tree to index the data set and prune unnecessary entries through the use of MBRs (Minimum Bounding Rectangles). However, as pointed out by [2, 4, 19], the use of R-tree or any other spatial indexes suffer from similar problems: When processing high-dimensional data sets, the performance declines to even worse than that of linear scan.

Figure 2 shows the comparison of performance between tree-based algorithms (BBR, MPA) and the simple scan (SIM, linear scan). According to the results, SIM outperforms these tree-based algorithms when processing RRQ in high dimensions. The reason for that inefficiency is that tree-based algorithms cannot divide data correctly in high dimensions, causing most of the MBRs to intersect with each other. Thus, even a small range query can overlap with a major proportion of the MBRs.

Figure 3 shows a geometric view of processing *RTK* queries. In this example, suppose that we treat p_4 as the query point q, then a line that crosses q is perpendicular to Tom's weight vector. The points in the gray area have a greater rank than q. Tree-based methodology filters entries that are entirely in the gray area and counts the number of points contained in filtered entries to record the rank of q. However, because q is within overlapping parts of MBRs, the tree-based algorithm cannot filter any parts of MBRs containing the Tom or Jerry's preferences. As a result, it has to go through most entries one by one and compute the scores. In these cases, traversal of the tree-based spatial index is not an efficient method.

For real-world applications, it is a natural requirement to process RRQ for high dimensional data (more than 3). Both the product's attributes and user's preferences are likely to be high-dimensional. For example, cell phones consumers care about many features, such as price, processor, storage, size, battery life, camera, etc. As another example, DIAN-



(a) Tom

(b) Jerry

Figure 3: Tree-base methodology processing *RTK* and search space (gray).

PING [1], a Chinese business-reviewing website, ranks restaurants by users' reviews on overall rate, food flavor, cost, service, environment, waiting time, etc. Therefore, processing RRQ with a high-dimensional data set is a significant problem, and due to the so-called "curse of dimensionality", simple scan offers a better performance than R-tree to solve it.

Despite its performances advantages on high-dimensional queries, there are challenges in processing RRQ with the simple scan. RRQ are more complicated queries than simple similarity searches such as the top-k query or the nearest neighbor search, and the time complexity of a naive simple scan method is $O(|P| \times |W|)$. RRQ require that every combination between P and W is checked before obtaining an answer. And this incurs a large number of pairwise computations. A comparison of 10K cell phones and 10K user preferences would necessitate $10K \times 10K = 100M$ computations. As a result, the enormous computational requirements cause the CPU cost to outweigh the I/O cost, which is the opposite of what happen in normal situations. We hold a preliminary experiment to confirm this by measuring the elapsed time for reading different sizes of data, for processing RRQ queries and for the pairwise computations in the inner product. Table 2 shows that the time taken to read different sizes of data file is almost negligible in the RRQ processing. Rather, the major cost of processing RRQ is the pairwise computations. We also found that the proportion of pairwise computations in processing RRQ grew from about 50% in 6-dimensional data to 90% in 100-dimensional data. In conclusion, in contrast to the usual strategy of saving I/O cost in other simple similarity searches, saving CPU computations is the key to process high-dimensional RRQ efficiently.

For the above reasons, we develop an optimized version of the simple scan, called the Grid-index algorithm (GIR) which reduces the amount of multiplication of inner product in the processing. First, We pre-compute some approximate multiplication values and store them into a 2d array named Grid-index. Then we pre-process the data P and W and create the approximate vectors $P^{(A)}$ and $W^{(A)}$ which indicate the index. In the GIR algorithm, we first scan the approximate vectors $P^{(A)}$ and $W^{(A)}$, then use them with the Grid-index to assemble upper and lower bounds, which help to filter most data without multiplications. After the filtering, we only need to refine few remaining data. In the worst case, it costs the I/O time for reading the $P^{(A)}$ and $W^{(A)}$, which is much less than original data and insignificant as concluded above.

Data size Elapsed time(ms)	1K	10K	100K
Reading data	5	26	146
Processing RRQ	240	9311	624318
−Pairwise computations	103	5321	352511

Table 2: Time cost for reading data and processing reverse rank queries with 6-dimensional data.

1.3 Contributions

The contributions of this paper are as follows:

- We elucidate that the simple scan is an appropriate way to process RRQ when processing high-dimensional data. We also demonstrate that CPU cost is the majority cost and that it is much larger than I/O processing. We are the first to conclude that a better approach for processing RRQ is to optimize the scan method.
- We propose a Grid-index, which uses pre-calculated score bounds to reduce multiplications in the simple scan. Based on Grid-index, we propose GIR algorithm which processes *RTK* and *RKR* queries more efficiently. Our method outperforms tree-based algorithms in almost all cases and all data sets, except for those in very low (less than 4) dimensional cases.
- We analyze the filter performance of tree-based algorithms and establish the GIR performance model. Theoretical analysis clarifies the limitation of the tree-based methods. The performance model of proposed GIR guarantees the efficiency of the Grid-index method is achieved at a negligible memory cost.

The rest of this paper is organized as follows: Section 2 summarizes the related work. Section 3 states the Grid-index concept and how to construct upper and lower bounds. In Section 4, we present the formal description of the GIR algorithm. Section 5 analyzes the performance of tree-based algorithms and gives a performance model for the Grid-index. Experimental results are shown in Section 6, and Section 7 concludes the paper.

2. RELATED WORK

For top-k queries, one possible approach to the top-k problem is the Onion technique [3]. This algorithm precomputes and stores convex hulls of data points in layers like an onion. The evaluation of a linear top-k query is accomplished by starting from the outermost layer and processing these layers inwardly. [8] proposed a system named PREFER that uses materialized views of top-k result sets that are very close to the scoring function in a query.

Reverse rank queries (RRQ) are the reverse version of the top-k queries. A typical query of RRQ is the reverse top-k query. [13, 14] introduced the reverse top-k query and presented two versions, namely monochromatic and bichromatic, and proposed a reverse top-k Threshold Algorithm (RTA). [5] indexed a dataset with a critical k-polygon for monochromatic reverse top-k queries in two dimensions. [17]

[1] http://www.dianping.com

propose a tree-base, branch-and-bound (BBR) algorithm which is the state-of-the-art approach for reverse top-k query. BBR indexes both data sets P and W in two R-trees, and points and weighting vectors are pruned through the branch-and-bound methodology. For applications, reverse top-k query was used in [16] to identified the most influential products, and in [15] to monitor the popularity of locations based on user mobility.

However, the reverse top-k query has a limitation that returns an empty result for an unpopular product. [22] introduced the reverse k-ranks query to ensure that any product in the data set can find their potential customers. Then proposed a tree-base algorithm named MPA (Marked Pruning Approach), which uses a d-dimensional histogram to index W and an R-tree to index P. Dong et al. [7] indicated that both reverse top-k and reverse k-rank queries were designed for only one product and cannot handle the product bundling. So they defined an aggregate reverse rank query that finds the top-k users for multiple query products.

Other works also considered a given data point and aimed at finding the queries that have this data point in their result set, such as the reverse (k) nearest neighbor (RNN or RKNN) [10, 20] that finds points that consider the query point as the nearest neighbor. RKNN may looks similar to RRQ, but they are actually very different. RKNN evaluates relative L_p distance in one Euclid space with between two certain points. On the other hand, RRQ focus on the absolute ranking value over all products, and the ranking scores are found through inner products of user preferences and products, from two different data spaces.

For other reverse queries, the reverse furthest neighbor (RFN) [21] and its extension RKFN (reverse k furthest neighbor) [18] find points that consider a query point as their furthest neighbor. The reverse skyline query uses the advantages of products to find potential customers based on the dominance of competitors products [6, 11]. However, reverse skyline query uses a desirable product data to describe the preference of a user. But in the definition of RRQ, the preference is described as a weighting vector.

For the space-partition tree-based structure, R*-tree [1], a variation on R-tree, improves pruning performance by reducing overlap in the tree construction. [9] used Hilbert space-filling curves to impose a linear ordering on the data rectangles in R-tree and improve the performance. [2] investigated and demonstrated the deficiencies of R-tree and R*-tree when dealing with high-dimensional data. As an improvement, a superior index structure named X-tree was proposed. X-tree uses a split algorithm to minimize overlap and utilizes the concept of super-nodes. In our opinion, X-tree can be seen as a middle approach between the R-tree and simple scan methods, because it uses the spatial tree structure to process the disjoint parts, and uses linear scan with the overlapping parts. For high-dimensional data, there are very few disjoint parts, causing there to be almost no advantage to the construction and look-up features of the X-tree.

It is well known that the overlapping nodes in high-dimensional space, is a shortcoming of tree structure. R. Weber *et al.* [19] proved that tree-based like [1, 2] is worse than linear scan in high-dimensional data and proposed a VAFILE filtering strategy. They divided the data space into buckets equally and use these buckets' upper and lower bounds to filter candidates. The goal of using VAFILE is to save I/O cost by scanning the bit-compressed file of buckets. However, we purpose to save the CPU computing in RRQ. [4] proposed a technique by "indexing the function" that pre-computing some key values of the L_p-distance function to avoid the expensive computing in high-dimensional nearest neighbour search.

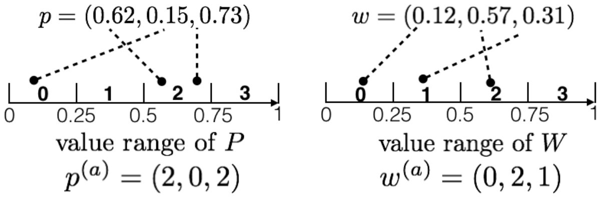


Figure 4: Equally dividing value range into 4 partitions, allocating real values into approximate intervals and getting the approximate vector $p^{(a)}$ and $w^{(a)}$.

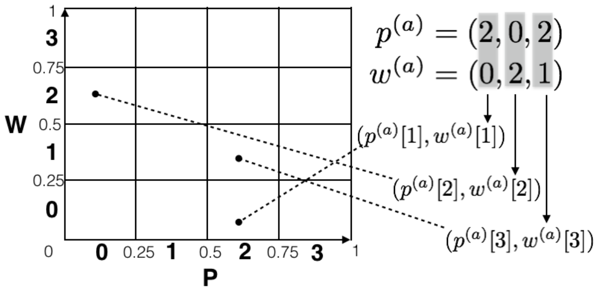


Figure 5: 4×4 Grids for points and weighting vectors, mapping $p^{(a)}$ and $w^{(a)}$ onto Grids.

3. GRID-INDEX

According the statement in Section 1.2, it stands to reason that using a simple scan with high-dimensional data is the most efficient approach. However, in this method, the multiplications of inner products take most of the processing time. We were inspired to study a method that could enhance the efficiency of the simple scan by avoiding multiplications for the inner product. In this section, we introduce the concept of Grid-index, which stores pre-calculated approximate multiplication values. The approximate values can form upper and lower bounds of a score and can be used in a filtering step for the simple scan approach.

3.1 Approximate Values in Grid-index

Concept of Grids. To confirm that the resultant score of the weighted sum function (inner product) is fair, all values in p must be in the same range, so must all values in w. We use this feature to allocate values into value ranges. As Figure 4 shows, in this example we partition the value range into 4 equal intervals. For the given $p = (0.62, 0.15, 0.73)$, the first attribute $p[1] = 0.62$ falls into the third partition $[0.5, 0.75]$. The second, $p[2] = 0.15$, falls into the first partition $[0, 0.25]$. We will store the partition numbers as an approximate vector, denoted as $p^{(a)}$ and $w^{(a)}$, so $p^{(a)} = (2, 0, 2)$ and $w^{(a)} = (0, 2, 1)$.

Since the inner product is the sum of pairwise multiplications of $p[i]$ and $w[i]$, we combine the ranges of p and w to form the grids. Figure 5 illustrates the 4 × 4 grids in this example. We can map an arbitrary pair of $(p[i], w[i])$ onto a certain grid, and different $(p[i], w[i])$ pairs may share the same grid location. The purpose of mapping the pairs onto the grid is to use the grids' corners to estimate the score of $p[i] \cdot w[i]$. By taking advantage of values having the same range, these grids can be re-used for mapping all pairs $(p[i], w[i])$, $i \in [1, d]$, $p \in P$ and $w \in W$.

Construction of Grid-index. Assume that we divide the value range of p and w into $n = 2^b$ partitions, and the position information of all elements in a vector are represented by a (n+1)-element vector α_p for points and α_w for weights. In the example of Figure 4, $\alpha_p = \alpha_w = (0, 0.25, 0.5, 0.75, 1)$. The Grid-index, denoted as *Grid*, is a 2-dimensional array and saves all multiplication results of all combinations between α_p and α_w:

$$Grid[i][j] = \alpha_p[i] \cdot \alpha_w[j],\ i, j \in [0, n] \tag{1}$$

Score Bounds and Precedence. According to the above Grid partition, we pre-store all approximate vectors for P and W, denoted as $P^{(A)}$ and $W^{(A)}$. The approximate vector $p^{(a)}$ for a given p is calculated by $p^{(a)}[i] = \lfloor p[i] \cdot n/r \rfloor$, where r is the range of $p[i]$'s attribute value. $w^{(a)}$ is calculated from w in the same way. Clearly, for a pair $(p[i], w[i])$ in the ith dimension, $Grid[p^{(a)}[i]][w^{(a)}[i]]$ is the lower bound and $Grid[p^{(a)}[i] + 1][w^{(a)}[i] + 1]$ is the upper bound. In the example, $p[1] = 0.62$, $w[1] = 0.12$ and $p^{(a)}[1] = 2$, $w^{(a)}[1] = 0$. Based on Equation (1), $Grid[2][0] = 0.5 \times 0$, $Grid[2{+}1][0{+}1] = 0.75 \times 0.25$, meaning $0.5 \times 0 \leq p[1] \cdot w[1] \leq 0.75 \times 0.25$.

For the inner product $f_w(p) = \sum_{i=1}^{d} p[i] \cdot w[i]$, based on properties of the inner product and features of the Grid-index, we know that:

$$L[f_w(p)] \leq f_w(p) \leq U[f_w(p)] \tag{2}$$

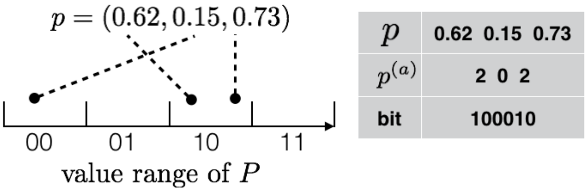
where $L[f_w(p)]$ and $U[f_w(p)]$, denoting the lower bound and the upper bound of $f_w(p)$, are given by

$$L[f_w(p)] = \sum_{i=1}^{d} Grid[p^{(a)}[i]][w^{(a)}[i]] \tag{3}$$

$$U[f_w(p)] = \sum_{i=1}^{d} Grid[p^{(a)}[i] + 1][w^{(a)}[i] + 1] \tag{4}$$

The relationship between p and q can be classified into three cases with the help of $L[f_w(p)]$ and $U[f_w(p)]$:

- Case 1 ($p \prec_w q$): If $U[f_w(p)] < f_w(q)$, p precedes q, p has a higher rank than q with w.
- Case 2 ($q \prec_w p$): If $L[f_w(p)] > f_w(q)$, q precedes p, p does not affect the rank of q with w.
- Case 3 ($p \asymp q$): Otherwise, p and q are incomparable, i.e., $L[f_w(p)] \leq f_w(q) \leq U[f_w(p)]$. The Grid-index cannot define whether p or q ranks higher with w.



p	0.62 0.15 0.73
$p^{(a)}$	2 0 2
bit	100010

Figure 6: 6-bit string for compressing the p to $p^{(a)}$.

Filtering Strategy. We scan the approximate vectors first, then use the Grid-index to obtain $L[f_w(p)]$ and $U[f_w(p)]$, and filter points that satisfy either Case 1 or Case 2 above. After scanning, if necessary, we carry out a refining phase, and compute the real score for all points in Case 3. Notice that throughout this process, we only calculated the sum and retrieved $L[f_w(p)]$ and $U[f_w(p)]$ of Equations (3) and (4). If a point p is in Case 1 or Case 2, we do not need to compute the real score $f_w(p)$, thus saving computational costs with multiplications to find the inner product.

3.2 Compress the Approximate Vectors

Storing all approximate vectors incurs extra storage costs for data sets P and W. To compress this storage, each approximate vector can be presented by a bit-string describing the interval which its elements fall. Figure 6 shows an example where the approximate vector $p^{(a)}$ is saved as a 6-bit string (100010), because 2 bits are needed to define 4 partitions for each of the 3 dimensions. Generally, if we divide the value range into 2^b partitions, then a ($b \times d$)-bit string is needed to store an approximate vector. According to the analysis in Section 5.3, $b = 6$ is enough for a good filtering performance. Usually, the original data is a 64-bit float value, so the storage overhead by the compressed 6-bit data is less than 1/10 of the original data [2]. This kind of bit-string compressing technique is also used in [19].

Reading approximate vectors with bit-string binary compression only has half the time costs compared to regular I/O operations. However, the superiority of I/O cost can be ignored because the CPU cost is far greater than the I/O cost in RRQ, as discussed in Section 1.2.

It may be argued that it would be the most efficient to store all the scores of each p and w directly. In reality, storing that amount of data is impossible due to the immense cost. For example, assume that there are $10K$ products and $10K$ weight vectors. For Grid-index, 20K tuples are needed to store the approximate vectors, but it would take $10K \times 10K = 100M$ tuples to store all the scores. The storage overhead for storing all scores is thousands of times of the approximate vectors in the proposed Grid-index method.

4. THE GIR ALGORITHM

Next, we use the Grid-index methodology to propose two versions of Grid-indexing algorithm for *RTK* and *RKR* queries. The two algorithms can be implemented easily by using the GInTop-k function that efficiently obtains the rank of query point q on a certain input w.

4.1 GInTop-k Function Based on Grid-index

Algorithm 1 describes the GInTop-k function based on Grid-index. GInTop-k scans each approximate vector $p_j^{(a)} \in P^{(A)} - Domin$. *Domin* is a global variable denoting a buffer

[2] When $n = 2^b$, then the storage cost for the approximate vectors are $|P^{(A)}| = \frac{b}{64}|P|$ and $|W^{(A)}| = \frac{b}{64}|W|$, if P and W's attributes are float values.

Algorithm 1 Grid-index checking q's rank (GInTop-k)

```
Require: $P^{(A)}, w_i^{(a)}, q, k, Grid, Domin$
Ensure: -1: discard $w_i$, $rnk$: include $w_i$
 1: $Cand \leftarrow \emptyset$
 2: $rnk \leftarrow Domin.size$
 3: for each $p_j^{(a)} \in P^{(A)} - Domin$ do
 4:    Calculate $U[f_w(p_j)]$ by Eq. (4)
 5:    if $U[f_{w_i}(p_j)] \leq f_{w_i}(q)$ then
 6:       $rnk$ ++          // $(p_j \prec_w q)$
 7:       if $p_j \prec q$ then
 8:          $Domin \leftarrow Domin \cup \{p_j\}$
 9:       if $rnk \geq k$ then
10:          return $-1$
11:    else
12:       Calculate $L[f_w(p_j)]$ by Eq. (3)
13:       if $L[f_{w_i}(p_j)] \leq f_{w_i}(q) \leq U[f_{w_i}(p_j)]$ then
14:          $Cand \leftarrow Cand \cup \{p_j\}$        // $(p_j \asymp q)$
15: Refine $Cand$ : compare real score and updating $rnk$.
16: if $rnk \geq k$ then
17:    return $-1$
18: else
19:    return $rnk$
```

recording dominating points. If p is in *Domin*, then every attribute of a point p is smaller than the corresponding attribute in q $(\forall p[i], i \in (0, d) : p[i] < q[i])$. Once q is given, points in *Domin* always rank better than q. During scanning, the number of points that rank better than q are counted by *rnk*, which is initialized by the size of *Domin* (line 2). The upper bound for the score is obtained using Grid-index (line 4). If the upper bound is smaller than the score of q (Case 1, line 5), then p_j must have a better rank than q for the weighting vector w_i, hence *rnk* increases by 1 (line 6). Anytime p_j is found dominating q, denoted by $p_j \prec q$, p_j will be appended to *Domin* (line 7-8). Whenever *rnk* reaches k (line 9), there are at least k points that rank better than q, thus the current w_i is not a result of *RTK* of q (-1 is returned). Otherwise, we get a lower bound from the Grid-index (line 12). If $q's$ score is between the lower bound and upper bound of p_j's score (in Case 3, line 13), then p_j is added to *Cand* for further refinement(line 14). After scanning $P^{(A)}$, if the algorithm did not return a decision, then a refinement step is necessary to establish (line 15). We check the original data of the points held in *Cand* and refine *rnk* in the same way, terminating immediately when it reaches k.

Computing $f_w(p)$ requires d multiplication operations and d addition operations. However, to find $U[f_w(p)]$ and $L[f_w(p)]$, it is only necessary to carry out d addition operations. Therefore, our approach will save d times of multiplication if $U[f_w(p)] \leq f_w(q)$ and the algorithm uses the branch at lines 5-10. When $U[f_w(p)] \geq f_w(q)$, our approach requires another d addition operations to find $L[f_w(p)]$, that is, an equivalent amount of additions to replace the multiplication operations in the evaluation of $f_w(p)$. In conclusion, using this method will save computational cost if any point can be filtered by the Grid-index. Section 5.3 proves that a low cost Grid-index can be used to filter over 99% of points.

4.2 Grid-index Algorithm

Now we introduce how Grid-index is applied to RRQ. Algorithm 2 and Algorithm 3 give the implementation of *RTK* and *RKR*.

For each approximate vector of $w_i^{(a)} \in W^{(A)}$, Algorithm 2 receives the result of filtering performed by GInTop-k (line 4). If the current w_i ranks q in its top-k, then w_i will be added into the result set of $RTOPk(q)$ (Line 5-6). If there exists more than k dominating points of q, the algorithm returns an empty set because q cannot be part of the top-k for any weighting vector w (line 7-8).

Unlike *RTK*, a heap structure of size k, denoted by *heap*, and a value *minRank* are introduced in Algorithm 3 for processing the *RKR*. For each $w_i^{(a)} \in W^{(A)}$, function GInTop-k is called first, *minRank* is passed to GInTop-k and used for filtering (line 5). If q ranks in the top-*minRank* (line 6), we insert w_i and *rnk* into the *heap*. The last rank of *heap* is pushed out after it holds more than k elements (line 7). Meanwhile, *minRank* is updated by the current last rank of *heap* (line 8). This ensures a self-refined bound and keeps the current k best results from W in *heap*. Finally, when the algorithm terminates, *heap* is returned as the result set.

Algorithm 2 Grid-index Reverse top-k (GIRTop-k)

```
Input: $P^{(A)}, W^{(A)}, q, k$
Output: $RTK$ result set $RTOPk(q)$
 1: create $Grid$ (Grid-index)
 2: $Domin \leftarrow \{\emptyset\}$
 3: for each $w_i^{(a)} \in W^{(A)}$ do
 4:    $rnk \leftarrow$ GInTop-$k(P^{(A)}, w_i^{(a)}, q, k, Grid, Domin)$
 5:    if $rnk \neq -1$ then
 6:       $RTOPk(q) \leftarrow RTOPk(q) \cup \{w_i\}$
 7:    if $Domin.size \geq k$ then
 8:       return $\{\emptyset\}$
 9: return $RTOPk(q)$
```

Algorithm 3 Grid-index Reverse k-ranks (GIRk-Rank)

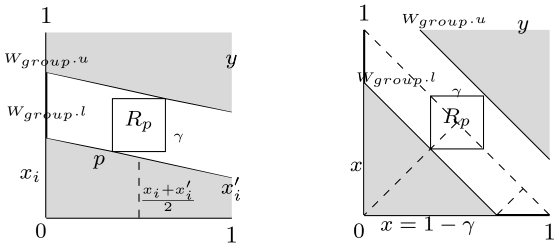
```
Input: $P^{(A)}, W^{(A)}, q, k$
Output: $heap = RKR$ result set
 1: create $Grid$ (Grid-index)
 2: $heap \leftarrow \{\emptyset\}$, $Domin \leftarrow \{\emptyset\}$
 3: $minRank \leftarrow \infty$
 4: for each $w_i^{(a)} \in W^{(A)}$ do
 5:    $rnk \leftarrow$ GInTop-$k(P^{(A)}, w_i^{(a)}, q,$ $minRank$, $Grid$,
       $Domin)$
 6:    if $rnk \neq -1$ then
 7:       $heap.insert$ $(w_i, rnk)$
 8:       $minRank \leftarrow heap$'s last rank.
 9: return $heap$
```

5. PERFORMANCE ANALYSIS

In this section, we first analyze the weakness of tree-based algorithms for RRQ. We then build a cost model for Grid-index that finds the ideal number of grids ($n \times n$), guaranteeing that specified filtering performance.

5.1 The Difficulty of Space-division in High Dimensional Data

We first observe the influence of the number of divisions through a space-division index. According to [22], MPA



(a) Trapezoidal prism

(b) Tetrahedron

Figure 7: Two kinds of Filtering areas (gray) of R-tree.

uses a d-dimensional histogram to group all weighting vectors W into *buckets*. Each dimension is partitioned into c equal-width intervals, in total, there are c^d *buckets*. As [22] suggests, $c = 5$, If $|W| = 100K$ with the 3-dimensional data, W is grouped in $5^3 = 125$ *buckets*. However, if $d = 10$, then there are $5^{10} \approx 9$ million *buckets*. It is not logical to filter only 100K weight vectors by testing the upper and lower bounds of such a huge number of *buckets*. In this case, scanning one by one would be more efficient.

5.2 Analysis of R-tree Filtering Performance

We test some range queries (within 1% area of the data space) over different d with an R-tree and observe the MBRs. Table 3 shows the average value of accessed MBRs' attributes. Not surprisingly, when $d > 6$, all (100%) of MBRs overlap in the query range, which means that all entries will be accessed during processing. As mentioned in Section 1.2, it is a shortcoming of tree-based algorithms that the MBRs will always overlap with each other when the data is high-dimensional.

Besides the shortcoming from the tree-based index itself, we also found that the filterable space of RRQ with tree-based methodology reduces as the dimensionality increases. This conclusion is supported by the following estimation.

Consider a tree-based algorithm that constructs an R-tree for the products P and assume that R_p is a MBR of this R-tree. In query processing, for each group of w's (denoted as W_{group}), points within R_p are checked. The upper and lower bounds of $f_{W_{group}}(R_p)$ are determined by the borders of W_{group} and R_p. As Figure 7 shows, The gray area is the safely filtered space. The shape of the gray area can be a hyper-prism, a hyper-tetra or a combination of the two. It means that in some of the dimensions (denoted as g) the area will be a triangle, while a trapezoid in others. Assume that the two kinds of shapes are separated clearly; then the proportion of filtered values can be obtained by measuring the volume:

$$Vol = Vol_{TetraX} \cdot Vol_{PrismX} + Vol_{TetraY} \cdot Vol_{PrismY} \quad (5)$$

To give an analytical result, we assume that R_p is in the centroid, so the two filtering areas are equal ($Vol_{TetraX} = Vol_{TetraY}$). Then the volume becomes

$$Vol = 2 \cdot Vol_{Tetra} \cdot Vol_{Prism} \quad (6)$$

Firstly, the volume of hyper-tetra is: [3]

$$Vol_{Tetra} = \frac{1}{g!}(\prod_{i=1}^{g} x_i) = \frac{1}{g!}(1-\gamma)^g \quad (7)$$

then, the volume of the hyper-prism (the area in Figure 7 (a)) is:

$$S_i = \frac{1}{2}(x_i + x_i') \cdot H \le (\frac{1-\gamma}{2}) \le \frac{1}{2} \quad (8)$$

where $H = 1$ is the length of the side. Imagine a 3 dimensional trapezoidal prism in the figure, the volume is:

$$Vol_{Prism3d} = \frac{1}{3}(S_1 + S_2 + \sqrt{S_1 S_2}) \cdot H \le \frac{1}{2} \quad (9)$$

This result holds for higher dimensional trapezoidal prisms. Consequently, the maximum volume gives the filtered area.

$$Vol_{max} = 2 \cdot \frac{1}{g!}(1-\gamma)^g \cdot \frac{1}{2} = \frac{1}{g!}(1-\gamma)^g \quad (10)$$

It is reasonable to assume that in half of the dimensions the filtered area is hyper-tetra in shape. We will consider a dataset of $d = 10$, $g = 5$, according to Equation (10), R-tree based methods can only filter at most $\frac{1}{5!}$ =0.8% of the data space.

This clearly shows that the space filtered by R-trees in RRQ becomes very small when encountering high-dimensional data. For all points in the space which can not be filtered, each $w[i] \cdot p[i]$ must be calculated and compared with that of the query point.

5.3 The Performance Model of Grid-index

To build a model of our Grid-index, we make the following assumption about the d-dimensional point data set: Values in all dimensions are independent of each other, and the sub-score in each dimension ($w[i] \cdot p[i]$) follows a uniform distribution. Both value ranges of P and W are divided into n partitions for the Grid-index.

Let the probability of a score S falling into a certain interval $(a,\ b)$ be $Prob(a < S < b)$, where $(a,\ b)$ is created by Grid-index. Data points with scores outside of $(a,\ b)$ can be filtered. We denote the filtering performance F by:

$$F(a,\ b) = 1 - Prob(a < S < b). \quad (11)$$

For example, if the probability of a point falling in an interval is 5%, then we say that the filter performance is 95%.

Obviously, $F(a,\ b)$ from Grid-index depends on the density of the grids ($n \times n$). More partitions n lead to smaller $Prob(a < S < b)$ and better filtering performance. However, larger n requires more memory, so it is important to find a suitable n that balances these factors. For this purpose, we first establish specific score properties and then define the relationship between F and n.

For the case of one dimension, dividing the range into equally n^2 partitions, the probability of a point p's score falling into a certain interval is obviously:

$$Prob(\frac{k}{n^2} < w \cdot p < \frac{k+1}{n^2}) = \frac{1}{n^2},\ k = 1, 2, ..., n^2. \quad (12)$$

[3] Recall that the area of a right triangle is $s = \frac{x_1 x_2}{2}$, and a tetrahedron has volume $v = \frac{x_3 s}{3} = \frac{x_1 x_2 x_3}{3 \cdot 2}$. if for (d-1) dim, the volume is $V_{d-1} \sim cx^{d-1}$ then $V_d = \int V_{d-1} dx \sim \frac{cx^d}{d}$.

Dimensionality	3	6	9	12	15	18	21	24
#MBR	1501	1480	1470	1470	1439	1479	1458	1456
diagonal length	4057.7	11744.3	19559.1	23807.9	31010.9	33717.1	36979.2	40515
Shape*	24.9	13.8	8.9	6.4	4.8	4.6	4.7	4.4
Overlaps in Query(1%)	30%	99.8%	100%	100%	100%	100%	100%	100%
Volume	$2.89 \times e^9$	$1.39 \times e^{21}$	$3.65 \times e^{33}$	$1.72 \times e^{45}$	$1.08 \times e^{58}$	$5.31 \times e^{69}$	$2.16 \times e^{81}$	$2.28 \times e^{93}$

* Shape is the ratio of the longest edge against the shortest one of an MBR.

Table 3: Observation of accessed MBRs of R-tree in query. 100K points indexed in R-tree, each MBR has 100 entries.

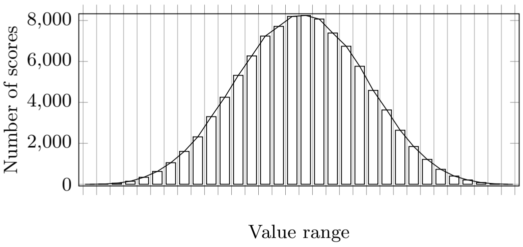


Figure 8: Grid-index scores distribution in dimension $d = 4$, partitions $n = 4$, $|P| = 100\text{K}$, $|W| = 100\text{K}$.

Now, we want to estimate the probability of p's score ($\sum_{i=1}^{d} w[i] \cdot p[i]$) falling in a score range obtained by Grid-index. For the discrete d dimension case:

$$Prob(\sum_{i=1}^{d}(w[i] \cdot p[i]) = s) \tag{13}$$

This probability can be found by the so called "Dice Problems": Rolling d n^2-sided dice and find the probability of obtaining s score. In this problem, a n^2-sided die corresponds to the score range of a single dimension which is equally partitioned in n^2 parts by Grid-index. The number of dice corresponds to the number of dimensions d, and the scores by rolling d dice becomes the point's score.

The number of ways obtaining score s is the coefficient of x^s in:

$$t(x) = (x^1 + x^2 + ... + x^{n^2})^d \tag{14}$$

By [12], the probability of obtaining s score on d n-sided dice is

$$Prob(s, d, n) = \frac{1}{n^{2d}} \sum_{k=0}^{\lfloor (s-d)/n^2 \rfloor} (-1)^k \binom{d}{k} \binom{s - n^2 k - 1}{d - 1} \tag{15}$$

The filtering performance of Grid-index can be presented by $1 - Prob(s, d, n)$. However, it is difficult to analyse the relationship between n and the filtering performance by Equation (15). On the other hand, we found that the distribution of scores approaches a normal distribution, even in low dimensional cases, such as 4. Figure 8 shows the observation of distribution of scores computed by Grid-index with $n = 4$ partitions, and the dimension $d = 4$. This encourages us to approximate the feature by normal distribution.

For a point p, $p[i] \cdot w[i]$ obeys a uniform distribution with range [0, r), average value μ and standard deviation σ, where

$$\mu = \frac{1}{2}r \qquad \sigma = \frac{1}{2\sqrt{3}}r \tag{16}$$

The average score value of a point p is

$$\overline{p \cdot w} = \frac{1}{d}\sum_{i=1}^{d}(p[i] \cdot w[i]) \tag{17}$$

By the central limit theorem, we have the following approximation when d is sufficiently large.

Lemma 1. *(Score Distribution). The following random variable*

$$Z = \frac{\sqrt{d}}{\sigma}(\overline{p \cdot w} - \mu) \tag{18}$$

follows the standard normal distribution (SND). In other words, $Z \sim N(0, 1)$, where μ and σ are as in Equation (16).

Note that $d \cdot \overline{p \cdot w}$ is the score of point p. Representing it by a random variable S, S follows a normal distribution with mean $\mu' = \mu d$ and standard deviation $\sigma' = \sigma\sqrt{d}$. By Equation (16),

$$\mu' = \frac{1}{2}rd \qquad \sigma' = \frac{\sqrt{d}}{2\sqrt{3}}r \tag{19}$$

From Lemma 1 and (11), we may now estimate the filtering performance.

Lemma 2. *(Filtering performance). The filtering performance of Grid-index, F, is given by*

$$\begin{aligned} F(x, x + \Delta) &= 1 - Prob(x < S < x + \Delta) \\ &= 1 - \int_{x}^{x+\Delta} f(x)dx \end{aligned} \tag{20}$$

where

$$f(x) = \frac{1}{\sigma'\sqrt{2\pi}} exp(-\frac{(x - \mu')^2}{2\sigma'^2}) \tag{21}$$

is the probability density function of $N(\mu',\ \sigma')$.

It is difficult to calculate the integral, but by rewriting Z in Lemma 1, The above equation can be:

$$Z = \frac{d \cdot \overline{p \cdot w} - \mu d}{\sigma\sqrt{d}} = \frac{S - \mu'}{\sigma'} \tag{22}$$

we can map S to $Z \sim N(0,\ 1)$ and need only to look up the SND table.

We are now ready to estimate the filtering performance of the Grid-index methodology. Recall that the score of a point is the sum of d addends. The score's range in each dimension is $[0, r)$, and it is equally divided into n^2 partitions. Thus, the value range computed by Grid-index of a d-dimensional points corresponds to range Δ:

$$\Delta = \frac{r}{n^2}d \tag{23}$$

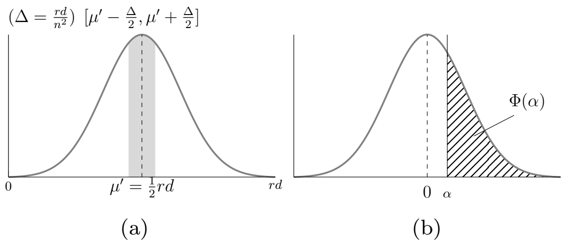


Figure 9: (a): The normal distribution of point scores $N(\mu', \sigma')$ and the largest probability interval (gray). (b): $\Phi(\cdot)$ of the SND showing $1 - \int_{-\alpha}^{\alpha} \cdot \; = 2\Phi(\alpha)$.

Our purpose is to find the number of partitions n which guarantees a certain filtering performance F in Lemma 2. To do this, it is sufficient to show the worst case. By Lemma 2, scores that fall within the interval illustrated by the gray part in Figure 9(a) which is located on either side of μ, have the largest probability and thus gives the worst F. Concentrating on this worst interval $[\mu' - \frac{\Delta}{2}, \mu' + \frac{\Delta}{2}]$, by Equation (22) and Equation (19), we find that $S_\Delta = \mu' \pm \frac{\Delta}{2}$ corresponds to

$$Z_\Delta = \frac{S_\Delta - \mu'}{\sigma'} = \frac{\mu' \pm \frac{\Delta}{2} - \mu'}{\sigma'} = \pm\frac{\sqrt{3d}}{n^2} \tag{24}$$

From Lemma 1, $Z \sim N(0,1)$, the filtering performance in the worst case can be given by

$$F(x, x+\Delta) > F_{worst}(x, x+\Delta) = 1 - \int_{\mu' - \frac{\Delta}{2}}^{\mu' + \frac{\Delta}{2}} f(x)dx = 2\Phi(\frac{\sqrt{3d}}{n^2}) \tag{25}$$

where $\Phi(\cdot)$ is the area shown in Figure 9 (b).

The above discussion leads to the following result.

Theorem 1. *Given $\epsilon < 1$, the filtering performance of n partitions is guaranteed to be above 1- ϵ in Grid-index such that*

$$n > \sqrt{\frac{2\sqrt{3d}}{\delta}} \tag{26}$$

where δ is determined by looking up the SND table at $(1 - \epsilon)/2$, that is,

$$\Phi(\frac{\delta}{2}) = \frac{1-\epsilon}{2} \tag{27}$$

Proof. *By Equation (26), $\frac{\delta}{2} > \frac{\sqrt{3d}}{n^2}$. Since Φ is a monotonically decreasing function (Figure 9), $\Phi(\frac{\sqrt{3d}}{n^2}) > \Phi(\frac{\delta}{2})$. Combining Equation (25) and Lemma 2, we have $F > 2\Phi(\frac{\delta}{2}) = 1 - \epsilon$* □

Example. To ensure that Grid-index filters out over 99% data, we set $\epsilon = 1\%$ ($\frac{(1-\epsilon)}{2} = 0.495$), thus the filtering performance is guaranteed to be better than $F_{worst}(\delta) = 99\%$. Looking up this value in the SND table, we have $\Phi(0.0125) = 0.495$, hence, $\delta = 0.025$. By Theorem 1, the sufficient number of partitions n is calculated by

$$\frac{\sqrt{3d}}{n^2} < \delta = 0.0125 \quad \longrightarrow n > \sqrt{\frac{2\sqrt{3d}}{\delta}} = \sqrt{80\sqrt{3d}} \tag{28}$$

P W	Uniform	Normal	Exponential
Uniform	99.3%	98.3%	99.0%
Normal	98.8%	96.5%	98.7%
Exponential	99.2%	97.5%	98.9%

Table 4: Filtering performance of Grid-index with different distributions. $|P| = 100K$, $|W| = 100K$, $d = 6$, $n = 32$

.

Parameter	Values
Data dimensionality d	$2 \sim 50$, **6**
Distribution of data set P	**UN**,CL,AC,RE
Data set cardinality $\|P\|$	50K,**100K**,1M,2M,5M
Distribution of data set W	**UN**,CL,RE
Data set cardinality $\|W\|$	50K,**100K**,1M,2M,5M
Experiment times	1000
Number of clusters	$\sqrt[3]{\|P\|}$, $\sqrt[3]{\|W\|}$
Variance σ_W^2,σ_P^2	0.1^2
Number of grids, n^2	4^2,8^2,16^2,32^2,64^2,$\mathbf{128^2}$
k (top-k and k-ranks)	**100**,200,300,400,500

Table 5: Experimental parameters and default values(in bold) .

If $d = 20$ then $n = 32$ satisfies Equation (28) hence a 32 × 32 Grid-index is enough for filtering over 99% data. The necessary memory is less than 8 K (32 × 32 × 8) Bytes.

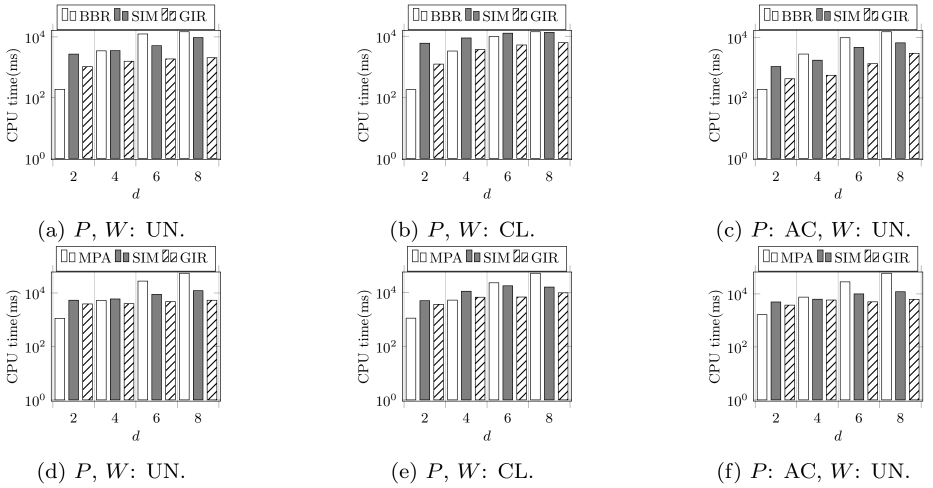
Theorem 1 is still true when $w[i] \cdot p[i]$ follows other distributions. The only difference is that a new μ_i and $\frac{\sigma_i}{\sqrt{d}}$ would have to be estimated, which would lead to a different partition n. We observed the filtering performance on some typical distributions, including the normal distribution ($\sigma = 10\%$) and exponential distribution ($\lambda = 2$). The filtering power of the Grid-index is shown in Table 4. Different σ between these distributions lead to slight differences in filtering power. But the filtering power is always efficient.

6. EXPERIMENT

In this section, we present the experimental evaluation. All algorithms are implemented in C++ and experiments are run on a Mac with a 2.6 GHz Intel Core i5 processor, 8GB RAM, 500GB flash storage space. We pre-read the R-tree, data sets P and W, approximated vectors P_A and W_A and the Grid-index into memory. According to Table 2, the I/O time is not relevant, so we focus on comparing our work only in terms of CPU processing time.

6.1 Experimental Setup

Data sets. For data set P, both real data (RE) and synthetic data sets are employed. Synthetic data sets are uniform (UN), anti-correlated (AC), and clustered (CL), with an attribute value range of $[0, 10K)$. The details on generating UN, AC, and CL data are in related research [13, 17]. To create weighting vectors W, there is additional UN and CL data that is generated in the same way. There are three real data sets, HOUSE, COLOR and DIANPING. HOUSE (Household) consists of 201,760 6-d tuples, representing the distribution percentages of an American family's annual payment on gas, electricity, water, heating, insurance and property tax. COLOR consists of 68,040 9-d tuples and describes features of images in the HSV color space. HOUSE and COLOR were also used in related works [13, 17]. DI-



(a) P, W: UN.

(b) P, W: CL.

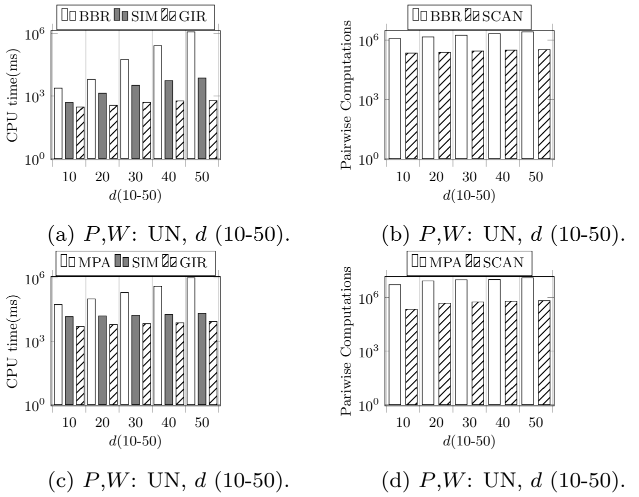
(c) P: AC, W: UN.

(d) P, W: UN.

(e) P, W: CL.

(f) P: AC, W: UN.

Figure 10: GIR vs BBR (a, b, c) for *RTK*, GIR vs MPA (d, e, f) for *RKR*. Performance on synthetic data with varying d (2-8), $|P| = |W| = 100$K, top-$k = 100$, k-ranks $= 100$, $n = 32$.



(a) P,W: UN, d (10-50).

(b) P,W: UN, d (10-50).

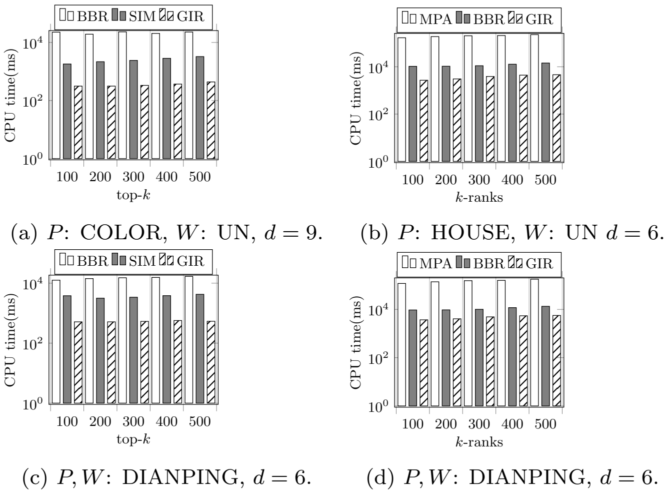
(c) P,W: UN, d (10-50).

(d) P,W: UN, d (10-50).

Figure 11: Performance on synthetic data with high dimensional d $(10-50)$, $|P| = |W| = 100$K, top-$k = 100$, k-ranks $= 100$, $n = 32$.

ANPING is a 6-d real world data set from a famous Chinese online business-reviewing website. It includes 3,605,300 reviews by 510,071 users on 209,132 restaurants about rate, food flavor, cost, service, environment and waiting time. We use the average scores of the reviews by the same user as his/her preference (w), and the average scores of the reviews on a restaurant as its attributes (p). RRQ can be anticipated to help to find target users for these restaurants.

Algorithms. We implemented BBR, MPA and Simple Scan algorithms (SIM). In BBR [17], both data sets P and W are indexed by R-tree, points and weighting vectors are pruned through the branch-and-bound methodology. MPA [22] uses an R-tree to index P and a d-dimensional histogram to group W in order to avoid checking every weighting vector. In SIM, for each w, all points in P are scanned and used to compute the score. SIM also maintains a *Domin* buffer to avoid unnecessary computing and terminates when current rank does not satisfy the conditions for *RTK* or *RKR*.



(a) P: COLOR, W: UN, $d = 9$.

(b) P: HOUSE, W: UN $d = 6$.

(c) P, W: DIANPING, $d = 6$.

(d) P, W: DIANPING, $d = 6$.

Figure 12: GIR vs Tree-base with RE data on varying "k", for *RTK* and *RKR* queries. $n = 32$.

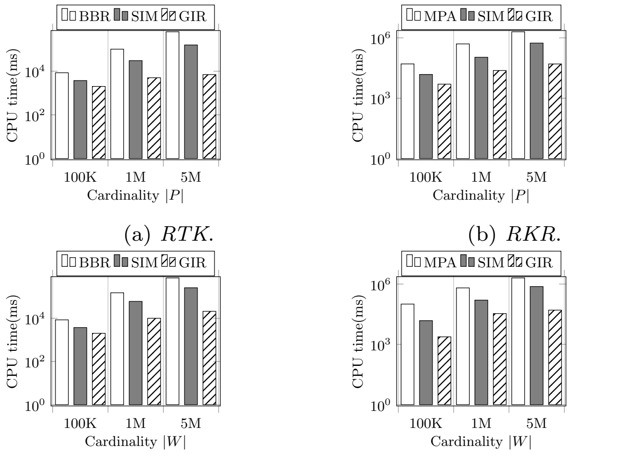
In conclusion, the only difference between SIM and GIR is that SIM computes a score for each p and w directly rather than using Grid-index for filtering.

Parameters. Parameters are shown in Table 5 where the default values are $d = 6$, $|P|$=100K, $|W|$=100K, k=100, the number of Grids is 32^2, and both P and W are UN data.

Metrics. We did each experiment over 1000 times, and present the average value. The query point q is randomly selected from P. Besides the query execution time required by each algorithm, we also observe the number of pairwise computations and the percentage of accessed data.

6.2 Experimental Results

Synthesis data with varying d. Figure 10 shows the performance of P (UN, AC, CL) and W (UN, CL) on synthetic data sets, with $|P|$=100K and $|W|$=100K, k=100, n = 32. Figures 10a, 10b and 10c show the CPU time and cost comparisons for *RTK* in low dimensions (2 to 8). GIR outperforms BBR in all distributions (UN,CL,AC) when data



(a) *RTK*.

(b) *RKR*.

(c) *RTK*.

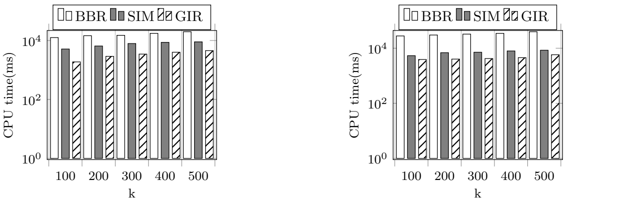
(d) *RKR*.

Figure 13: Scalability for all algorithms with varying $|P|$ (a,b) and $|W|$ (c,d), top-k = 100, k-ranks = 100, n = 32, d = 6.

has over 4 dimensions. SIM outperforms BBR when data has more than 6 dimensions, with the exception of CL data, since R-tree can group and prune more points when the data set is clustered. GIR always exceeds SIM at least 2 times because GIR uses score bounds from Grid-index to skip most data without doing multiplications. The results of *RKR* are shown in Figures 10d, 10e, 10f, GIR outperforms simple scan SIM at all times and outperforms the tree-based MPA with 4 to 8-dimensional data.

In high-dimensions (10-50), as shown in Figures 11a and 11c, the query time taken by tree-based method increases rapidly for the two reasons we presented in Sections 1.2 and 5.2: overlapping MBRs and little space to prune. Figure 11b, 11d present the number of pairwise computations for all algorithms, both BBR and MPA use more computations than the simple scan. Notice that the computation numbers for GIR and SIM are equal and are both titled "SCAN" in the figures. On the other hand, GIR is the most stable method and only grows slightly. This confirms that GIR is only slightly affected by increasing dimensionality.

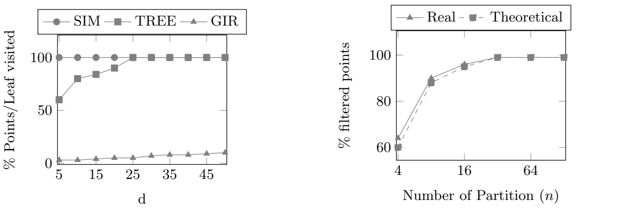
Real data with varying "k". For the performance of these algorithms on real data sets (RE) with varying k. Notice that k has a different meaning, it is a query condition in *RTK* and a result size in *RKR*. Figures 12a, 12b show the results from data set HOUSE and COLOR, and data set W is generated as UN data. We process COLOR with *RTK* and HOUSE with *RKR*. Clearly, GIR is consistently superior to tree-based algorithms (BBR and MPA) and SIM, though all are stable for various k values. For the DIANPING dataset, P and W contain the average scores vectors from the reviews of users and restaurants. We peform *RTK* and *RKR* queries on DIANPING data and the Figures 12c and 12d show the comparison results. As we expected, the GIR algorithm is the most efficient for this real-world application data set.



(a) *RTK*.

(b) *RKR*.

Figure 14: GIR vs Tree-base with varying k, for *RTK* and *RTK* queries. P, W: UN, n = 32, d = 6.



(a) Visited data, n = 32.

(b) Filtered data, d = 20.

Figure 15: (a) Visited data for all algorithms on varying d, (b) Filtering data (%) of Grid-index on varying n. $|P|$ = 100K, $|W|$ = 100K. P, W: UN.

Scalability with varying $|P|$ and $|W|$. According Figure 13, as the cardinality of data set increases P (Figures 13a and 13b) or W (Figures 13c and 13d), GIR becomes significantly superior to tree-based algorithms (BBR, MPA) and SIM. n = 32 is sufficient to filter more than 99% of points for a 6-d dataset based our Theorem 1. Thus, the CPU cost increased only slightly as the scale increased.

Effect on "k". Figures 12, 14 also show the performance changes when k increases from 100 to 500. All algorithms are insensitive to k because $k \ll |P|$ and $k \ll |W|$.

Accessed data points. Figure 15a shows the percentage of visited data in the leaf nodes of the R-tree and original data points on UN data. As predicted by our analysis, R-tree degenerated to a simple scan through all leaf nodes with high-dimensional data. However, GIR accesses a relatively small amount of data after filtering with Grid-index.

Effect on value range partitions n. Figure 15b shows the percent of 20-d data which can be filtered with Grid-index with various Grid numbers ($n \times n$). We created Grid-index with different n from 4 to 128 and observed the filtering of data points. The results confirm the analytical result guaranteed by Theorem 1. n = 32 is enough to guarantee a high Grid-index efficiency.

7. CONCLUSION

Reverse rank queries are useful in many applications. In marketing analysis, they can be used to help manufacturers recognize their consumer base by matching their product features with user preferences. The state-of-the-art approaches for both reverse top-k (BBR) and reverse k-ranks (MPA) are tree-based algorithms, and are not designed to deal with high-dimensional data. In this paper, we proposed the Grid-index and the GIR algorithm to overcome the cost of high-dimensional computing when processing reverse rank queries. Theoretical analysis and experimental results confirmed the efficiency of the proposed algorithm when compared to the tree-based algorithms especially in high-dimensional cases.

In future work, there are two extensions for GIR algorithm. The first is to find a heuristic method to adapt GIR to different data distributions by using non-equal-width Grid-index. This is easy to implement by merging and splitting some grids of the equal-width Grid-index based on the distributions of the given P and W. The challenging point is the model of filtering performance with varied distributions in different dimensions. The second extension is to do optimization when the user preferences data $w \in W$ has many zero entry, i.e., when W is sparse. Since in practice, a user is normally interested in a few attributes of the products.

Acknowledgement

This work was partly supported by JSPS KAKENHI Grant-in-Aid for Scientific Research (B) (Grant number: 26280037).

8. REFERENCES

[1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990.*, pages 322–331, 1990.

[2] S. Berchtold, D. A. Keim, and H. Kriegel. The x-tree : An index structure for high-dimensional data. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 28–39, 1996.

[3] Y. Chang, L. D. Bergman, V. Castelli, C. Li, M. Lo, and J. R. Smith. The onion technique: Indexing for linear optimization queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 391–402, 2000.

[4] H. Chen, J. Liu, K. Furuse, J. X. Yu, and N. Ohbo. Indexing expensive functions for efficient multi-dimensional similarity search. *Knowl. Inf. Syst.*, 27(2):165–192, 2011.

[5] S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides. Indexing reverse top-k queries in two dimensions. In *Database Systems for Advanced Applications, 18th International Conference, DASFAA 2013, Wuhan, China, April 22-25, 2013. Proceedings, Part I*, pages 201–208, 2013.

[6] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 291–302, 2007.

[7] Y. Dong, H. Chen, K. Furuse, and H. Kitagawa. Aggregate reverse rank queries. In *Database and Expert Systems Applications - 27th International Conference, DEXA 2016, Porto, Portugal, September 5-8, 2016, Proceedings, Part II*, pages 87–101, 2016.

[8] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 259–270, 2001.

[9] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 500–509, 1994.

[10] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 201–212, 2000.

[11] X. Lian and L. Chen. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 213–226, 2008.

[12] J. V. Uspensky. *Introduction to Mathematical Probability.* New York: McGraw-Hill, 1937.

[13] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 365–376, 2010.

[14] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Monochromatic and bichromatic reverse top-k queries. *IEEE Trans. Knowl. Data Eng.*, 23(8):1215–1229, 2011.

[15] A. Vlachou, C. Doulkeridis, and K. Nørvåg. Monitoring reverse top-k queries over mobile devices. In *Proceedings of the Tenth ACM International Workshop on Data Engineering for Wireless and Mobile Access, MobiDE 2011, Athens, Greece, June 12, 2011*, pages 17–24, 2011.

[16] 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.

[17] A. Vlachou, C. Doulkeridis, K. Nørvåg, and Y. Kotidis. Branch-and-bound algorithm for reverse top-k queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 481–492, 2013.

[18] S. Wang, M. A. Cheema, X. Lin, Y. Zhang, and D. Liu. Efficiently computing reverse k furthest neighbors. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 1110–1121, 2016.

[19] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 194–205, 1998.

[20] S. Yang, M. A. Cheema, X. Lin, and Y. Zhang. SLICE: reviving regions-based pruning for reverse k nearest neighbors queries. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 760–771, 2014.

[21] B. Yao, F. Li, and P. Kumar. Reverse furthest neighbors in spatial databases. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 664–675, 2009.

[22] Z. Zhang, C. Jin, and Q. Kang. Reverse k-ranks query. *PVLDB*, 7(10):785–796, 2014.

Parallel and Distributed Processing of Spatial Preference Queries using Keywords

Christos Doulkeridis[1], Akrivi Vlachou[1], Dimitris Mpestas[1,2], Nikos Mamoulis[2]
[1]Dept. of Digital Systems, University of Piraeus, Greece
[2]Dept. of Computer Science & Engineering, University of Ioannina, Greece
cdoulk@unipi.gr, avlachou@aueb.gr, dmpestas@teemail.gr, nikos@cs.uoi.gr

ABSTRACT

Advanced queries that combine spatial constraints with textual relevance to retrieve objects of interest have attracted increased attention recently due to the ever-increasing rate of user-generated spatio-textual data. Motivated by this trend, in this paper, we study the novel problem of parallel and distributed processing of *spatial preference queries using keywords*, where the input data is stored in a distributed way. Given a set of keywords, a set of spatial data objects and a set of spatial feature objects that are additionally annotated with textual descriptions, the *spatial preference query using keywords* retrieves the top-k spatial data objects ranked according to the textual relevance of feature objects in their vicinity. This query type is processing-intensive, especially for large datasets, since any data objects may belong to the result set while the spatial range defines the score, and the k data objects with the highest score need to be retrieved. Our solution has two notable features: (a) we propose a deliberate re-partitioning mechanism of input data to servers, which allows parallelized processing, thus establishing the foundations for a scalable query processing algorithm, and (b) we boost the query processing performance in each partition by introducing an early termination mechanism that delivers the correct result by only examining few data objects. Capitalizing on this, we implement parallel algorithms that solve the problem in the MapReduce framework. Our experimental study using both real and synthetic data in a cluster of sixteen physical machines demonstrates the efficiency of our solution.

1. INTRODUCTION

With the advent of modern applications that record the position of mobile users by means of GPS, and the extensive use of mobile smartphones, we have entered the era of Big Spatial Data. The fact that an increasing amount of user-generated content (e.g., messages in Twitter, photos in Flickr, etc.) is geotagged also contributes to the daily creation of huge volumes of location-based data. Apart from spatial locations, the data typically contain textual descriptions or annotations. Analyzing and exploiting such textually annotated location-based data is estimated to bring high economic benefits in the near future.

In order to extract useful insights from this wealth of Big Spatial Data, advanced querying mechanisms are required that retrieve results of interest from massively distributed spatio-textual data. In this paper, we study an advanced query type that retrieves data objects based on the textual relevance of other (feature) objects in their spatial neighborhood. In particular, given a keyword-based query, a set of spatial *data objects* and a set of spatial *feature objects* that are additionally annotated with textual descriptions, the *spatial preference query using keywords* retrieves the top-k spatial data objects ranked according to the textual relevance of feature objects in their vicinity. This query is generic, as it can be used to retrieve locations of interest based on the relevance of Tweets in their vicinity, based on popular places (bars, restaurants, etc.), and/or based on the comments of other people in the surrounding area.

However, processing this query raises significant challenges. First, due to the query definition, every data object is a potential result and cannot be pruned by spatial or textual constraints. Second, the distribution of data raises the need to find a way to parallelize computation by assigning units of work that can be processed independently from others. In this paper, we address these technical challenges and provide the first solution to parallel/distributed processing of the spatial preference query using keywords. Our approach has two notable features: (a) we propose a method to parallelize processing by deliberately re-partitioning input data, in such a way that the partitions can be processed in parallel, independently from each other, and (b) within each partition, we apply an early termination mechanism that eagerly restricts the number of objects that need to be processed in order to provide the correct result set.

In more detail, we make the following contributions in this paper:

- We formulate and address a novel problem, namely parallel/distributed evaluation of spatial preference queries using keywords over massive and distributed spatio-textual data.
- We propose a grid-based partitioning method that uses careful duplication of feature objects in selected neighboring cells and allows independent processing of subsets of input data in parallel, thus establishing the foundations for a scalable, parallel query evaluation algorithm.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.29

- We further boost the performance of our algorithm by introducing an early termination mechanism for each independent work unit, thereby reducing the processing cost.

- We demonstrate the efficiency of our algorithms by means of experimental evaluation using both real and synthetic datasets in a medium-sized cluster.

The remainder of this paper is structured as follows: Section 2 presents the preliminary concepts and necessary background. Section 3 formally defines the problem under study and explains the rationale of our approach along with a brief overview. The proposed query processing algorithm that relies on the grid-based partitioning is presented in Section 4. Section 5 presents the algorithms that use early termination. Section 6 analyzes the complexity of the proposed algorithms. Then, in Section 7, we present the results of our experimental evaluation. Related research efforts are outlined in Section 8. Finally, in Section 9, we provide concluding remarks.

2. PRELIMINARIES

In this section we give a brief overview of MapReduce and HDFS, and define the type of queries we will focus on.

2.1 MapReduce and HDFS

Hadoop is an open-source implementation of MapReduce [4], providing an environment for large-scale, fault-tolerant data processing. Hadoop consists of two main parts: the HDFS distributed file system and MapReduce for distributed processing.

Files in HDFS are split into a number of large blocks which are stored on DataNodes, and one file is typically distributed over a number of DataNodes in order to facilitate high bandwidth during parallel processing. In addition, blocks can be replicated to multiple DataNodes (by default three replicas), in order to ensure fault-tolerance. A separate NameNode is responsible for keeping track of the location of files, blocks, and replicas thereof. HDFS is designed for use-cases where large datasets are loaded ("write-once") and processed by many different queries that perform various data analysis tasks ("read-many").

A task to be performed using the MapReduce framework has to be specified as two steps. The *Map* step as specified by a map function takes some input (typically from HDFS files), possibly performs some computation on this input, and redistributes it to worker nodes (a process known as "shuffle"). An important aspect of MapReduce is that both the input and output of the Map step is represented as key-value pairs, and that pairs with same key will be processed as one group by a Reducer. As such, the *Reduce* step receives all values associated with a given key, from multiple map functions, as a result of the re-distribution process, and typically performs some aggregation on the values, as specified by a reduce function.

It is important to note that one can customize the redistribution of data to Reducers by implementing a *Partitioner* that operates on the output key of the map function, thus practically enforcing an application-specific grouping of data in the Reduce phase. Also, the ordering of values in the reduce function can be specified, by implementing a customized Comparator. In our work, we employ such customizations to obtain a scalable and efficient solution to our problem.

Symbol	**Description**
O	Set of data objects
p	Object in O, $p \in O$
F	Set of feature objects
f	Feature object in F, $f \in F$
$f.\mathcal{W}$	Keywords associated with feature object f
$q(k, r, \mathcal{W})$	Query for top-k data objects
$w(f, q)$	Textual relevance of feature f to query q
$\overline{w}(f, q)$	Upper bound of $w(f, q)$
$dist(p, f)$	Spatial distance between p and f
$\tau(p)$	Score of data object p
$\overline{\tau}$	Score of the k-th best data object
R	Number of Reduce tasks
$\mathcal{C} = \{C_1, \ldots, C_R\}$	Grid cells

Table 1: Overview of symbols.

2.2 Spatial Preference Queries

The spatial preference query belongs to a class of queries that rank objects based on the quality of other (feature) objects in their spatial neighborhood [12, 16, 17]. Inherently a spatial preference query assumes that two types of objects exist: the data objects, which will be ranked and returned by the query, and the feature objects, which are responsible for ranking the data objects. As such, the feature objects determine the score of each data object according to a user-specified metric. Spatial preference queries find more applications in the case of textually annotated feature objects [14], where the score of data objects is determined by a textual similarity function applied on query keywords and textual annotations of feature objects. This query is known as top-k spatio-textual preference query [14]. In this paper, we study a distributed variant of this query.

3. PROBLEM STATEMENT AND OVERVIEW OF SOLUTION

3.1 Problem Formulation

Consider an *object* dataset O of spatial objects $p \in O$, where p is described by its coordinates $p.x$ and $p.y$. Also, consider a *feature* dataset F of spatio-textual objects $f \in F$, which are represented by spatial coordinates $f.x$ and $f.y$, and a set of keywords $f.\mathcal{W}$.

The spatial preference query using keywords returns the k data objects $\{p_1, \ldots, p_k\}$ from O with the highest score. The score of a data object $p \in O$ is defined by the scores of feature objects $f \in F$ in its spatial neighborhood. As already mentioned, each feature object f is associated with a set of keywords $f.\mathcal{W}$. A query q consists of a neighborhood distance threshold r, a set of query keywords $q.\mathcal{W}$ for the feature set F, and the value k that determines how many data objects need to be retrieved. For a quick overview of the basic symbols used in this paper, we refer to Table 1.

Given a query $q(k, r, \mathcal{W})$ and a feature object $f \in F$, we define the *non-spatial score* $w(f, q)$ that indicates the goodness (quality) of f as the similarity of sets $q.\mathcal{W}$ and $t.\mathcal{W}$. In this work, we employ Jaccard similarity for this purpose. Obviously, the domain of values of $w(f, q)$ is the range $[0, 1]$.

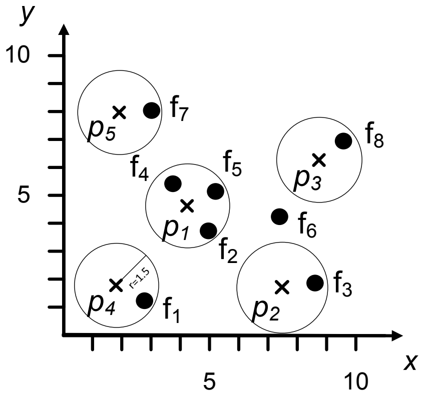


Figure 1: Example of spatial preference query using keywords (SPQ).

Object	X	Y	Keywords	Jaccard
p_1	4.6	4.8	-	-
p_2	7.5	1.7	-	-
p_3	8.9	5.2	-	-
p_4	1.8	1.8	-	-
p_5	1.9	9.0	-	-
f_1	2.8	1.2	italian,gourmet	0.5
f_2	5.0	3.8	chinese,cheap	0
f_3	8.7	1.9	sushi,wine	0
f_4	3.8	5.5	italian	1
f_5	5.2	5.1	mexican,exotic	0
f_6	7.4	5.4	greek,traditional	notInRange
f_7	3.0	8.1	italian,spaghetti	0.5
f_8	9.5	7.0	indian	0

Table 2: Example of datasets and scores for query $q.\mathcal{W} = \{italian\}$.

DEFINITION 1. *(*Non-spatial score $w(f, q)$*): Given a query q and a feature object $f \in F$, the non-spatial score $w(f, q)$ determines the textual relevance between the set of query keywords $q.\mathcal{W}$ and the keywords $f.\mathcal{W}$ of f using Jaccard similarity:*

$$w(f, q) = \frac{|q.\mathcal{W} \cap f.\mathcal{W}|}{|q.\mathcal{W} \cup f.\mathcal{W}|}$$

The score $\tau(p)$ of a data object p is determined by the feature objects that are within distance r from p. More specifically, $\tau(p)$ is defined by the maximum non-spatial score $w(f, q)$ of any feature object f in the r-neighborhood of p. This range-based neighborhood condition is typically used in the related work [12, 14, 16, 17]. Formally,

DEFINITION 2. *The score $\tau(p)$ of p based on feature dataset F, given the range-based neighborhood condition r is defined as:*

$$\tau(p) = max\{w(f, q) \mid f \in F : d(p, f) \leq r\}$$

EXAMPLE 1. *Figure 1 depicts an example. The spatial area contains both data objects (denoted as p_i) and feature objects (denoted as f_i). The data objects represent hotels and the feature objects represent restaurants. Assume a user issues the following query: Find the best (top-k) hotels that have an Italian restaurant nearby. Let us assume that k=1 and "nearby" is translated to $r = 1.5$ units of distance. Then, the query is expressed as: Find the top-1 data object for which a highly ranked feature object exists based on the keyword "italian" and at a distance of at most 1.5 units.*

Table 2 lists the locations and descriptions of both data and feature objects. Only feature objects f_1, f_4 and f_7 have a common term with the user specified query (the keyword "italian"). Thus, only f_1, f_4 and f_7 will have a Jaccard score other than 0. In the last column of Table 2 the Jaccard score for all feature objects is shown. The score of each data object is influenced only by the feature objects within a distance of at most 1.5 units. In Figure 1 the circles with radius 1.5 range units and center each data object include the feature objects that are nearby each data object and influence its score. The actual score of a data object is the highest score of all nearby feature objects. Data object p_4 has a score of 0.5 due to feature object f_1, data object p_1 has a score of 1 because of feature object f_4 and data object p_5 has a score of 0.5 due to feature object f_7. Hence, the top-1 result is object p_1.

In the parallel and distributed setting that is targeted in this paper, datasets O and F are horizontally partitioned and distributed to different machines (servers), which means that each server stores only a fraction (partition) of the entire datasets. In other words, a number of partitions $O_i \in O$ and $F_i \in F$ of datasets O and F respectively exists, such that $\bigcup O_i = O$, $O_i \bigcap O_j = \emptyset$ for $i \neq j$, and $\bigcup F_i = F$, $F_i \bigcap F_j = \emptyset$ for $i \neq j$. Due to horizontal partitioning, any (data or feature) object belongs to a single partition (O_i or F_i respectively). We make no assumption on the number of such partitions nor on having equal number of data and feature object partitions. Also, no assumptions are made on the specific partitioning method used; in fact, our proposed solution is independent of the actual partitioning method employed, which makes it applicable in the most generic case.

PROBLEM 1. *(*Parallel/Distributed Spatial Preference Query using Keywords (SPQ)*) Given an object dataset O and a feature dataset F, which are horizontally partitioned and distributed to a set of servers, the parallel/distributed spatial preference query using keywords returns the k data objects $\{p_1, \ldots, p_k\}$ from O with the highest $\tau(p_i)$ scores.*

3.2 Design Rationale

The spatial preference query using keywords (SPQ) targeted in this paper is a complex query operator, since any data object p may belong to the result set and the spatial range cannot be used for pruning the data space. As a result, the computation becomes more challenging and efficient query processing mechanisms are required that can exploit parallelism and the availability of hardware resources. Parallelizing this query is also challenging because any given data object p and all feature objects within the query range r from p must be assigned to the same server to ensure the correct computation of the score $\tau(p)$ of p. As such, a repartitioning mechanism is required in order to assign (data and feature) objects to servers in a deliberate way that allows local processing at each server. To achieve the desired independent score computation at each server, duplication of feature objects to multiple servers is typically necessary.

Based on this, we set the following objectives for achieving parallel, scalable and efficient query processing:

- **Objective #1:** parallelize processing by breaking the work into independent parts, while minimizing feature object duplication. In addition, the union of the results in each part should suffice to produce the final result set.
- **Objective #2:** avoid processing the input data in its entirety, by providing early termination mechanisms for query processing.

To meet the above objectives, we design our solution by using the following two techniques:

- a grid-partitioning of the spatial data space that uses careful duplication of feature objects in selected neighboring cells, in order to create independent work units (Section 4), and
- sorted access to the feature objects in a deliberate order along with a thresholding mechanism that allows early termination of query processing that guarantees the correctness of the result (Section 5).

4. GRID-BASED PARTITIONING AND INITIAL ALGORITHM

In this section, we present an algorithm for solving the parallel/distributed spatial preference query using keywords, which relies on a grid-based partitioning of the 2-dimensional space in order to identify subsets of the original data that can be processed in parallel. To ensure correctness of the result computed in parallel, we re-partition the input data to grid cells and deliberately duplicate some feature objects in neighboring grid cells. As a result, this technique lays the foundation for parallelizing the query processing and leads to the first scalable solution.

4.1 Grid-based Partitioning

Consider a regular, uniform grid in the 2-dimensional data space that consists of R cells: $\mathcal{C} = \{C_1, \ldots, C_R\}$. Our approach assigns all data and feature objects to cells of this grid, and expects each cell to be processed independently of the other cells. In MapReduce terms, we assign each cell to a single processing task (Reducer), thus all data that is mapped to this cell need to be sent to the assigned Reducer.

The re-partitioning mechanism operates in the following way. Based on its spatial location, an object (data or feature object) is assigned to the cell that encloses it in a straightforward way. However, some feature objects must be additionally assigned to other cells (i.e., duplicated), in order to ensure that the data in each cell can indeed be processed independently of the other cells and produce the correct result. More specifically, given a feature object $f \in C_j$ and any grid cell C_i ($C_i \neq C_j$), we denote by $MINDIST(f, C_i)$ the minimum distance between feature object f and C_i. This distance is defined as the distance of f to the nearest edge of C_i, since f is outside C_i. When this minimum distance is smaller than the query radius r, i.e., $MINDIST(f, C_i) \leq r$, then it is possible that f is within distance r from a data object $p \in C_i$. Therefore, f needs to be assigned (duplicated) also to cell C_i. The following lemma guarantees the correctness of the afore-described technique.

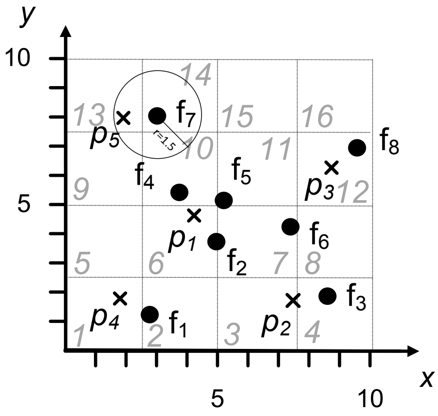


Figure 2: Example of grid partitioning.

LEMMA 1. *(Correctness) Given a parallel/distributed spatial preference query using keywords with radius r, any feature object $f \in C_j$ must be assigned to all other grid cells $C_i (C_i \neq C_j)$, if $MINDIST(f, C_i) \leq r$.*

Figure 2 illustrates the same dataset as in Figure 1 and a 4x4 grid (the numbering of the cells is shown in the figure). Consider a query with radius $r = 1.5$, and let us examine feature object f_7 as an example. Assuming that f_7 has at least one common term in its keyword set ($f_7.\mathcal{W}$) with the user specified query ($q.\mathcal{W}$), then f_7 may affect neighboring cells located near cell with identifier C_{14}. It is fairly easy to see that f_7 needs to be duplicated to cells C_9, C_{10}, and C_{13}, for which $MINDIST(f_7, C_i) \leq r$, thus the score of data objects located in these cells may be determined by f_7.

Before presenting the algorithm that exploits this grid partitioning, we make a note on how to select an appropriate grid size, as this affects the amount of duplication required. It should also be noted that in our approach the grid is defined at query time, after the value of r is known. Let α denote the length of the edge of a grid cell. For now, we should ensure that $\alpha \geq r$, otherwise excessive replication to neighboring cells would be performed. Later, in Section 6, we provide a thorough analysis on the effect of the grid cell size to the amount of duplicated data.

4.2 Parallel Algorithm

We design a parallel algorithm, termed *pSPQ*, that solves the problem in MapReduce. The Map phase is responsible for re-partitioning the input data based on the grid introduced earlier. Then, in the Reduce phase, the problem of reporting the top-k data objects is solved in each cell independently of the rest. This is the part of the query that dominates the processing time; the final result is produced by merging the k results of each of the R cells and returning the top-k with the highest score. However, this last step can be performed in a centralized way without significant overhead, given that the number of these results is small because k is typically small.

In more detail, in the Map phase, each Map task (Mapper) receives as input some data objects and some feature objects, without any assumptions on their location. Each Mapper is responsible for assigning data and feature objects to grid cells, including duplicating feature objects. Each grid cell

Algorithm 1 *pSPQ*: Map Function

```
1: Input: $q(k, r, \mathcal{W})$, grid cells $\mathcal{C} = \{C_1, \ldots, C_R\}$
2: function $MAP(x$: input object)
3: $C_i \leftarrow \{C_i : C_i \in \mathcal{C}$ and $x$ enclosed in $C_i\}$
4: if $x$ is a data object then
5:     $x.tag \leftarrow 0$
6:     output $\langle(i, x.tag), x\rangle$
7: else
8:     $x.tag \leftarrow 1$
9:     if $(x.\mathcal{W} \cap q.\mathcal{W} \neq \emptyset)$ then
10:        output $\langle(i, x.tag), x\rangle$
11:        for $(C_j \in \mathcal{C}$, such that $MINDIST(x, C_j) \leq r)$ do
12:            output $\langle(j, x.tag), x\rangle$
13:        end for
14:    end if
15: end if
16: end function
```

corresponds to a single Reduce task, which will take as input all objects assigned to the respective grid cell. Then, the Reducer can accurately compute the score of any data object located in the particular grid cell and report the top-k.

4.2.1 Map Phase

Algorithm 1 shows the pseudo-code of the Map phase, where each call of the Map function processes a single object denoted by x, which can be a data object or a feature object. First, in line 3, the cell C_i that encloses object x is determined. Then, if x is a data object, it is tagged ($x.tag$) with the value 0, otherwise with the value 1. In case of a data object, x is output using a *composite key* that consists of the cell id i and the tag as key, and as value the entire data object x. In case of a feature object, we apply a simple pruning rule (line 9) to eliminate feature objects that do not affect the result of the query. This rule practically eliminates from further processing any feature object that has no common keyword with the query keywords, i.e., $q.\mathcal{W} \cap f.\mathcal{W} = \emptyset$. The reason is that such feature objects cannot contribute to the score of any data object, based on the definition of our query. This pruning rule can significantly limit the number of feature objects that need to be sent to the Reduce phase. For the remaining feature objects that have at least one common keyword with the query, they are first output with the same composite key as above, and value the entire feature object x. In addition, we identify neighboring cells C_j that comply with Lemma 1, and replicate the feature object in those cells too. In this way, we have partitioned the initial data to grid cells and have performed the necessary duplication of feature objects.

The output key-values of the Map phase are grouped by cell id and assigned to Reduce tasks using a customized Partitioner. Also, in each Reduce task, we order the objects within each group by their tag, so that data objects precede feature objects. This is achieved through the use of the composite keys for sorting. As a result, it is guaranteed that each Reducer accesses the feature objects after it has accessed all data objects.

4.2.2 Reduce Phase

Algorithm 2 *pSPQ*: Reduce Function

```
1: Input: $q(k, r, \mathcal{W})$
2: function $REDUCE(key$, $V$: objects assigned to cell with id $key$)
3: $L_k \leftarrow \emptyset$
4: for $(x \in V)$ do
5:     if $x$ is a data object then
6:         Load $x$ in memory $O_i$
7:         $score(x) \leftarrow 0$ // initial score
8:     else
9:         if $w(x, q) > \overline{\tau}$ then
10:            for $(p \in O_i)$ do
11:                if $d(p, x) \leq r$ then
12:                    $score(p) \leftarrow \max\{score(p), w(x, q)\}$
13:                    update list $L_k$ of top-$k$ data objects and $\overline{\tau}$
14:                end if
15:            end for
16:        end if
17:    end if
18: end for
19: for $p \in L_k$ do
20:     output $\langle p, score(p)\rangle$ // at this point:$score(p) = \tau(p)$
21: end for
22: end function
```

As already mentioned, a Reduce task processes all the data assigned to a single cell and reports the top-k data objects within the respective cell. The pseudo-code of the Reduce function is depicted in Algorithm 2. First, all data objects are accessed one-by-one and loaded in memory (O_i). Moreover, a sorted list L_k of the k data objects p_i with higher scores $\tau(p_i)$ is maintained. Let $\overline{\tau}$ denote the k-th best score of any data object so far. Then, for each feature object x accessed, its non-spatial score $w(x, q)$ (i.e., textual similarity to the query terms) is compared to $\overline{\tau}$. Only if the non-spatial score $w(x, q)$ is larger than $\overline{\tau}$ (line 9), may the top-k list of data objects be updated. Therefore, in this case we test all combinations of x with the data objects p kept in memory O_i. If such a combination (x, p) is within distance r (line 11), then we check if the temporary score of p denoted by $score(p)$ can be improved based on x (i.e., $w(x, q)$), and if that is the case we check whether p has obtained a score that places it in the current top-k list of data objects (L_k). Line 12 shows how the score can be improved, however we omit from the pseudo-code the check of score improvement of p for sake of simplicity. Then, in line 13, the list L_k is updated. As explained, this update is needed only if the score of p is improved. In this case, if p already exists in L_k we only update its score, otherwise p is inserted into L_k. After all feature objects have been processed, L_k contains the top-k data objects of this cell.

4.2.3 Limitations

The above algorithm provides a correct solution to the problem in a parallel manner, thus achieving Objective #1. However, in each Reducer, it needs to process the entire set of feature objects in order to produce the correct result. In the following section, we present techniques that overcome this limitation, thereby achieving significant performance gains.

5. ALGORITHMS WITH EARLY TERMINATION

Even though the technique outlined in the previous section

Algorithm 3 *eSPQlen*: Map Function (Section 5.1)

```
1:  Input: $q(k, r, \mathcal{W})$, grid cells $\mathcal{C} = \{C_1, \ldots, C_R\}$
2:  function $MAP(x$: input object)
3:  $C_i \leftarrow \{C_i : C_i \in \mathcal{C}$ and $x$ enclosed in $C_i\}$
4:  if $x$ is a data object then
5:      output $\langle (i, 0), x \rangle$
6:  else
7:      if $(x.\mathcal{W} \cap q.\mathcal{W} \neq \emptyset)$ then
8:          output $\langle (i, |x.\mathcal{W}|), x \rangle$
9:          for ($C_j \in \mathcal{C}$, such that $MINDIST(x, C_j) \leq r$) do
10:             output $\langle (j, |x.\mathcal{W}|), x \rangle$
11:         end for
12:     end if
13: end if
14: end function
```

enables parallel processing of independent partitions to solve the problem, it cannot guarantee good performance since it requires processing both data partitions in a cell in their entirety (including duplicated feature objects). To alleviate this shortcoming, we introduce two alternative techniques that achieve *early termination*, i.e., report the correct result after accessing all data objects but only few feature objects. This is achieved by imposing a deliberate order for accessing feature objects in each cell, which in turn allows determining an upper bound for the score of any unseen feature object. When this upper bound cannot improve the score of the current top-k object, we can safely terminate processing of a given Reducer.

5.1 Accessing Feature Objects by Increasing Keyword Length

The first algorithm that employs early termination, termed *eSPQlen*, is based on the intuition that feature objects f with long textual descriptions that consist of many keywords ($|f.\mathcal{W}|$) are expected to produce low scores $w(f, q)$. This is due to the Jaccard similarity used in the definition of $w(f, q)$ (Defn. 1), which has $|q.\mathcal{W} \bigcup f.\mathcal{W}|$ in the denominator. Based on this, we impose an ordering of feature objects in each Reducer by increasing keyword length, aiming at examining first feature objects that will produce high score values $w(f, q)$ with higher probability.

In more details, given the keywords $q.\mathcal{W}$ of a query q, and a feature object f with keywords $f.\mathcal{W}$, we define a bound for the best possible Jaccard score that this feature object can achieve as:

$$\overline{w}(f, q) = \begin{cases} 1 & , |f.\mathcal{W}| < |q.\mathcal{W}| \\ \frac{|q.\mathcal{W}|}{|f.\mathcal{W}|} & , |f.\mathcal{W}| \geq |q.\mathcal{W}| \end{cases} \quad (1)$$

Given that feature objects are accessed by increased keyword length, this bound is derived as follows. As long as feature objects f are accessed that satisfy $|f.\mathcal{W}| < |q.\mathcal{W}|$, it is not possible to terminate processing, thus the bound takes the value of 1. The reason is that when $|f.\mathcal{W}| < |q.\mathcal{W}|$ holds, it is possible that a subsequent feature object f' with more keywords than f may have higher Jaccard score than f. However, as soon as it holds that $|f.\mathcal{W}| \geq |q.\mathcal{W}|$ the bound (best possible score) equals:

$$\frac{\min\{|q.\mathcal{W}|, |f.\mathcal{W}|\}}{\min\{|q.\mathcal{W}|, |f.\mathcal{W}|\} + |f.\mathcal{W}| - |q.\mathcal{W}|} = \frac{|q.\mathcal{W}|}{|f.\mathcal{W}|}$$

because in the best case the intersection of sets $q.\mathcal{W}$ and $f.\mathcal{W}$ will be equal to: $\min\{|q.\mathcal{W}|, |f.\mathcal{W}|\}$, while their union will be equal to: $\min\{|q.\mathcal{W}|, |f.\mathcal{W}|\} + |f.\mathcal{W}| - |q.\mathcal{W}|$.

Algorithm 4 *eSPQlen*: Reduce Function with Early Termination (Section 5.1)

```
Input: $q(k, r, \mathcal{W})$
function $REDUCE(key, V$: objects assigned to cell with id $key)$
$L_k \leftarrow \emptyset$
for $(x \in V)$ do
    if $x$ is a data object then
        Load $x$ in memory $O_i$
        $score(x) \leftarrow 0$ // initial score
    else
        if $\overline{\tau} \geq \overline{w}(x, q)$ then
            break
        end if
        if $w(x, q) > \overline{\tau}$ then
            for $(p \in O_i)$ do
                if $d(p, x) \leq r$ then
                    $score(p) \leftarrow \max\{score(p), w(x, q)\}$
                    update list $L_k$ of top-$k$ data objects and $\overline{\tau}$
                end if
            end for
        end if
    end if
end for
for $p \in L_k$ do
    output $\langle p, score(p) \rangle$ // at this point:$score(p) = \tau(p)$
end for
end function
```

Recall that $\overline{\tau}$ denotes the k-th best score of any data object so far. Then, the condition for early termination during processing of feature objects by increasing keyword length, can be stated as follows:

Lemma 2. *(Correctness of Early Termination* eSPQlen*) Given a query q and an ordering of feature objects based on increasing number of keywords, it is safe to stop accessing more feature objects as soon as a feature object f is accessed with:*

$$\overline{\tau} \geq \overline{w}(f, q)$$

Based on this analysis, we introduce a new algorithm that follows the paradigm of Section 4, but imposes the desired access order to feature objects and is able to terminate early in the Reduce phase.

5.1.1 Map Phase

Algorithm 3 describes the Map phase of the new algorithm. The main difference to the algorithm described in Section 4 is in the use of the composite key when objects are output by the Map function (lines 8 and 10). The composite key contains two parts. The first part is the cell id, as previously, but the second part is a number. The second part corresponds to the value zero in the case of data objects, while it corresponds to the length $|f.\mathcal{W}|$ of the keyword description in the case of a feature object f. The rationale behind the use of this composite key is that the cell id is going to be used to group objects to Reducers, while the second part of the key is going to be used to establish the

Algorithm 5 *eSPQsco*: Map Function (Section 5.2)

```
1: Input: $q(k, r, \mathcal{W})$, grid cells $\mathcal{C} = \{C_1, \ldots, C_R\}$
2: function $MAP(x$: input object)
3: $C_i \leftarrow \{C_i : C_i \in \mathcal{C}$ and $x$ enclosed in $C_i\}$
4: if $x$ is a data object then
5:    output $\langle (i, 2), x \rangle$
6: else
7:    if $(x.\mathcal{W} \cap q.\mathcal{W} \neq \emptyset)$ then
8:       output $\langle (i, w(x, q)), x \rangle$
9:       for ($C_j \in \mathcal{C}$, such that $MINDIST(x, C_j) \leq r$) do
10:         output $\langle (j, w(x, q)), x \rangle$
11:      end for
12:   end if
13: end if
14: end function
```

ordering in the Reduce phase in increased order of the number used. In this sorted order, data objects again precede feature objects, due to the use of the zero value. Between two feature objects, the one with the smallest length of keyword description (i.e., fewer keywords) precedes the other in the sorted order.

5.1.2 Reduce Phase

As already mentioned, feature objects with long keyword lists are expected to result in decreased textual similarity (in terms of Jaccard value). Thus, our hope is that after accessing feature objects with few keywords, we will find a feature object that has so many keywords that all remaining feature objects in the ordering cannot surpass the score of k-th best data object thus far.

Algorithm 4 explains the details of our approach. Again, only the set of data objects assigned to this Reducer is maintained in memory, along with a sorted list L_k of the k data objects with best scores found thus far in the algorithm. The condition for early termination is based on the score $\overline{\tau}$ of the k-th object in list L_k and the best potential score $\overline{w}(f, q)$ of the current feature object f (line 9).

5.2 Accessing Feature Objects by Decreasing Score

In this section, we introduce an even better early termination algorithm, termed *eSPQsco*. The rationale of this algorithm is to compute the Jaccard score in the Map phase and use this score as second part of the composite key. In essence, this can enforce a sorted order in the Reduce phase where feature objects are accessed from the highest scoring feature object to the lowest scoring one.

To explain the intuition of the algorithm, consider the feature object with highest score. Any data object located within distance r from this feature object is guaranteed to belong to the result set, as no other data object can acquire a higher score. This observation leads to a more efficient algorithm that can (in principle) produce results when accessing even a single feature object. As a result, the algorithm is expected to terminate earlier, by accessing only a handful of feature objects. This approach incurs additional processing cost at the Map phase (i.e., computation of the Jaccard score), but the imposed overhead to the overall execution time is minimal.

Algorithm 6 *eSPQsco*: Reduce Function with Early Termination (Section 5.2)

```
Input: $q(k, r, \mathcal{W})$
function $REDUCE(key$, $V$: objects assigned to cell with id $key$)
for $(x \in V)$ do
   if $x$ is a data object then
      Load $x$ in memory $O_i$
   else
      if $\exists p \in O_i : d(p, x) \leq r$ then
         output $\langle p, w(x, q) \rangle$ // here: $w(x, q) = \tau(p)$
         $cnt++$
         if $(cnt = k)$ then
            break
         end if
      end if
   end if
end for
end function
```

LEMMA 3. *(Correctness of Early Termination* eSPQsco*)* *Given a query q and an ordering of feature objects $\{f_i\}$ based on decreasing score $w(f_i, q)$, it is safe to stop accessing more feature objects as soon as k data objects are retrieved within distance r from any already accessed feature object.*

5.2.1 Map Phase

Algorithm 5 describes the Map function, where the only modifications are related to the second part of the composite key (lines 5, 8, and 10). In the case of data objects, this must be set to a value strictly higher than any potential Jaccard value, i.e., it can be set equal to 2, since the Jaccard score is bounded in the range $[0, 1]$. Thus, data objects will be accessed before any feature object. In the case of a feature object f, it is set to the Jaccard score $w(f, q)$ of f with respect to the query q. Obviously, the customized Comparator must also be changed in order to enforce the ordering, from highest scores to lowest scores.

5.2.2 Reduce Phase

Algorithm 6 details the operation of the Reduce phase. After all data objects are loaded in memory, the feature objects are accessed in decreasing order of their Jaccard score to the query. For each such feature object f, any data object located within distance r is immediately reported as a result within the specific cell. As soon as k data objects have been reported as results, the algorithm can safely terminate.

6. THEORETICAL RESULTS

In this section, we analyze the space and time complexity in the Reduce phase, which relate to the number of cells and the number of duplicate objects.

6.1 Complexity Analysis

Let R denote the number of Reducers, which is also equivalent to the number of grid cells. Further, let O_i and F_i denote the subset of the data and feature objects assigned to the i-th Reducer respectively. Notice that F_i contains both the feature objects enclosed in the cell corresponding to the i-th Reducer, as well as the duplicated feature objects that are located in other neighboring cells. In other words, it holds that $\bigcup_{i=1}^{R} |F_i| \geq |F|$.

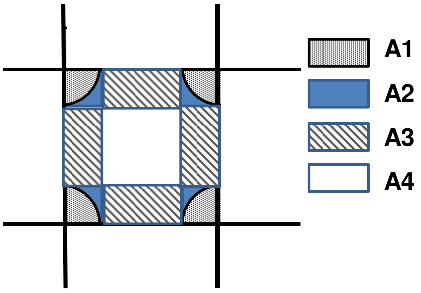


Figure 3: Breaking a cell in areas based on the number of duplicates.

In the case of the initial parallel algorithm that does not use early termination (Section 4), a Reducer needs to store in memory the data objects $|O_i|$ and the list of k data objects with best scores, leading to space complexity: $O(|O_i|)$ since $|O_i| >> k$. On the other hand, the time complexity is: $O(|O_i| \cdot |F_i|)$, since in worst case for all data objects and for each feature object the score is computed. In practice, when using the early termination the processing cost of each Reducer is significantly smaller, since only few feature objects need to be examined before reporting the correct result set.

If we make the simplistic assumption that the work is shared fairly in the R Reducers (e.g., assuming uniform distribution and a regular uniform grid), then we can replace in the above formulas: $|O_i| = \frac{|O|}{R}$. Let us also consider the *duplication factor* d_f of the feature dataset F, which is a real number that is grid-dependent and data-dependent, such that: $\bigcup_{i=1}^{R} |F_i| = d_f \cdot |F|$. Then, we can also replace in the above formulas: $|F_i| = \frac{d_f \cdot |F|}{R}$. Thus, the processing cost of a Reducer is proportional to: $|O_i| \cdot |F_i| = \frac{|O|}{R} \cdot \frac{d_f \cdot |F|}{R}$.

6.2 Estimation of the Duplication Factor

In the following, we assume that the size a of each side of a grid cell is larger than twice the query radius, i.e., $a \geq 2r$, or equivalently $r \leq \frac{a}{2}$. This is reasonable, since we expect r to be smaller than the size of a grid cell.

Depending on the area where a feature object is positioned, different number of duplicates of this object will be created. Figure 3 shows an example of a grid cell. Given a feature object enclosed into a cell, we identify four different cases. If the feature object has a distance smaller than or equal to r from any cell corner then the feature object is enclosed in the area A_1 that is depicted as the dotted area. In this case, the feature object must be duplicated to all three neighboring cells to the corner of the cell. If the feature object has a distance smaller than or equal to r from two cell borders but not from a cell corner then the feature object is enclosed in area A_2. This area is depicted with solid blue color defined by the four rectangles, but does not include the circles. If located in A_2, then only 2 duplicates will be created (not on the diagonal cell). The third area is A_3 depicted as dashed area and corresponds to the feature object that have a distance smaller than or equal to r from only one border of the cell. In this case, only one duplicate is needed. Finally, if the feature object is enclosed in the remaining area of the cell (white area, called area A_4), no duplication is needed.

Obviously, since it holds $r \leq \frac{a}{2}$ any feature object that belongs to a cell is located in only one of these four areas. Let $|A_i|$ denote the surface of area A_i, and A denote the area of the complete cell. Then:

- $|A_1| = 4 \cdot \frac{\pi r^2}{4} = \pi r^2$
- $|A_2| = 4 \cdot r^2 - |A_1| = (4 - \pi) r^2$
- $|A_3| = 4 \cdot (a - 2r) r$
- $|A_4| = a^2 - |A_1| - |A_2| - |A_3| = (a - 2r)^2$
- $|A| = a^2$

Let $P(A_i)$ denote the probability that a feature object belongs to area A_i. Then, if we assume uniform distribution of feature objects in the space, we obtain the following probabilities: $P(A_i) = \frac{|A_i|}{|A|}$. Based on this, given n feature objects enclosed in a cell, we can calculate the total number of feature objects (including duplicates), denoted by $\widehat{n}$, of the n feature points:

$$\widehat{n} = 3 \cdot n \cdot P(A_1) + 2 \cdot n \cdot P(A_2) + n \cdot P(A_3) + n$$

and we can calculate the duplication factor d_f for this cell:

$$d_f = \frac{\widehat{n}}{n} =$$
$$3 \cdot P(A_1) + 2 \cdot P(A_2) + P(A_3) + 1 =$$
$$3\frac{\pi r^2}{a^2} + 2\frac{(4-\pi)r^2}{a^2} + \frac{4 \cdot (a - 2 \cdot r) r}{a^2} + 1 =$$
$$\frac{\pi r^2}{a^2} + \frac{4 \cdot r}{a} + 1$$

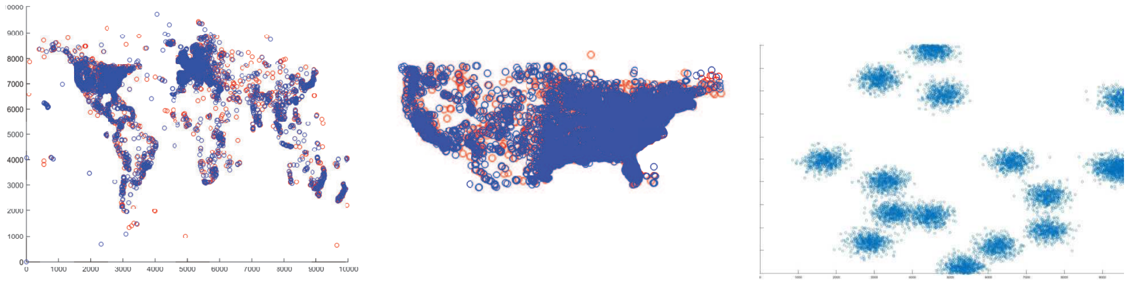
Based on the above formula, we conclude that the worst value of d_f is $3 + \frac{\pi}{4}$ for the case of $a = 2 \cdot r$ and it holds that $1 \leq d_f \leq 3 + \frac{\pi}{4}$ for any query range r such that $a \leq 2 \cdot r$. Also, the duplication factor depends only on the ratio of the cell size to the query range, under the assumption of uniform distribution. Moreover, the formula shows that smaller cell size α (compared to the query range r) increases the number of duplicated feature objects. Put differently, a larger cell size α reduces the duplication of feature objects.

6.3 Analysis of the Cell Size

Even though using a larger cell size α reduces the total number of feature objects, it also has significant disadvantages. First, it results in fewer cells thus reducing parallelism. Second, the probability of obtaining imbalanced partitions in the case of skewed data is increased. Let us assume that all R cells can be processed in a single round, i.e., the hardware resources are sufficient to launch R Reduce tasks in parallel[1]. In this case, the total processing time depends on the performance of one Reducer, which as mentioned before depends on $|O_i| \cdot |F_i| = \frac{|O|}{R} \cdot \frac{d_f \cdot |F|}{R} = |O| \cdot |F| \cdot \frac{d_f}{R^2}$. If we normalize the dataset in $[0, 1] \times [0, 1]$, then $\alpha \leq 1$ and $R = \frac{1}{a}$. Then, $|O_i| \cdot |F_i| = |O| \cdot |F| \cdot d_f \cdot a^4$. In order to study the performance of one Reducer while varying a, it is sufficient to study $d_f \cdot a^4$ since the remaining factors are constant.

Based on the estimation of d_f in the previous section, $d_f \cdot a^4 = (\frac{\pi r^2}{a^2} + \frac{4 \cdot r}{a} + 1) \cdot a^4 = \pi \cdot r^2 \cdot a^2 + 4 \cdot r \cdot a^3 + a^4$. If we consider r as a constant, then for increasing positive values of a, the value of the previous equation increases, which means that the complexity of the algorithm increases. Thus, a smaller cell size α increases the number of cells

[1] We make this assumption to simplify the subsequent analysis, but obviously the number of cores can be smaller than the number of grid cells, and in this case a Reducer will process multiple cells.



(a) Flickr (FL). (b) Twitter (TW). (c) Clustered (CL).

Figure 4: Illustration of spatial distribution of datasets.

and parallelism, and also reduces the processing cost of each Reducer.

7. EXPERIMENTAL EVALUATION

In this section, we report the results of our experimental study. All algorithms are implemented in Java.

7.1 Experimental Setup

Platform. We deployed our algorithms in an in-house CDH cluster consisting of sixteen (16) server nodes. Each of the nodes d1-d8 has 32GB of RAM, 2 disks for HDFS (5TB in total) and 2 CPUs with a total of 8 cores running at 2.6 GHz. The nodes d9-d12 have 128GB of RAM, 4 disks for HDFS (8TB in total) and 2 CPUs with a total of 12 cores (24 hyperthreads) running at 2.6 GHz. Finally, each of the nodes d13-d16 is equipped with 128GB RAM, 4 disks for HDFS (8TB in total), and 2 CPUs with a total of 16 cores running at 2.6GHz. Each of the servers in the cluster function as DataNode and NodeManager, while one of them in addition functions as NameNode and ResourceManager. Each node runs Ubuntu 12.04. We use the CDH 5.4.8.1 version of Cloudera and Oracle Java 1.7. The JVM heap size is set to 1GB for Map and Reduce tasks. We configure HDFS with 128MB block size and replication factor of 3.

Datasets. In order to evaluate the performance of our algorithms we used four different large-scale datasets. Two real datasets are included, a dataset of tweets obtained from Twitter and a dataset of images obtained from Flickr. The Twitter dataset (TW) was created by extracting approximately 80 million tweets which requires 5.7GB on disk. Besides a spatial location, each tweet contains several keywords extracted from its text, with 9.8 keywords on average per tweet, while the size of the dictionary is 88,706 keywords. The Flickr dataset (FL) contains metadata of approximately 40 million images, thus capturing 3.5GB on disk. The average number of keywords per image is 7.9 and the dictionary contains 34,716 unique keywords.

In addition, we created two synthetic datasets in order to test the scalability of our algorithms with even larger datasets. The first synthetic dataset consists of 512 million spatial (data and feature) objects that follow a uniform (UN) distribution in the data space. Each feature object is assigned with a random number of keywords between 10 and 100, and these keywords are selected from a vocabulary of size 1,000. The total size of the file is 160GB. The second synthetic dataset follows a clustered (CL) distribution. We generate 16 clusters whose position in space is selected at random. All other parameters are the same. The total size of the generated dataset is 160GB, as in the case of UN. Figure 4 depicts the spatial distribution of the datasets employed in our experimental study. In all cases, we randomly select half of the objects to act as data objects and the other half as feature objects.

Parameter	**Values**

Datasets	Real: {TW, FL}
	Synthetic: {UN, CL}

Query keywords ($\|q.\mathcal{W}\|$)	1, **3**, 5, 10
Query radius (r) (% of side α of grid cell)	5%, **10%**, 25%, 50%
Top-k	5, **10**, 50, 100
Grid size (FL, TW)	35x35, **50x50**, 75x75, 100x100
Grid size (UN, CL)	10x10, **15x15**, 50x50, 100x100

Table 3: Experimental parameters (default values in bold).

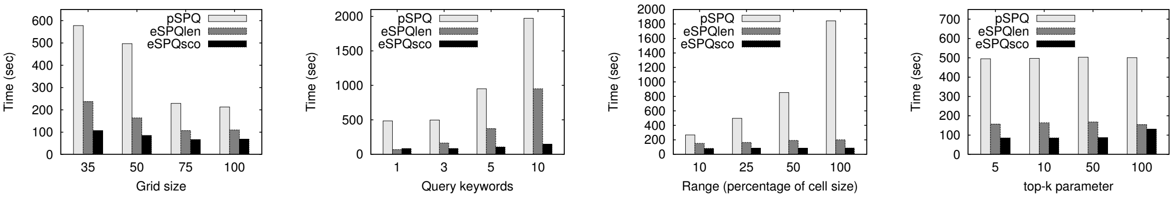
Algorithms. We compare the performance of the following algorithms that are used to compute the spatial preference query using keywords in a distributed and parallel way in Hadoop:

- *pSPQ*: the parallel grid-based algorithm without early termination (Section 4),
- *eSPQlen*: the parallel algorithm that uses early termination by accessing feature objects based on increasing keyword length (Section 5.1), and
- *eSPQsco*: the parallel algorithm that uses early termination by accessing feature objects based on decreasing score (Section 5.2).

For clarification purposes, we note that centralized processing of this query type is infeasible in practice, due to the size of the underlying datasets and the time necessary to build centralized index structures.

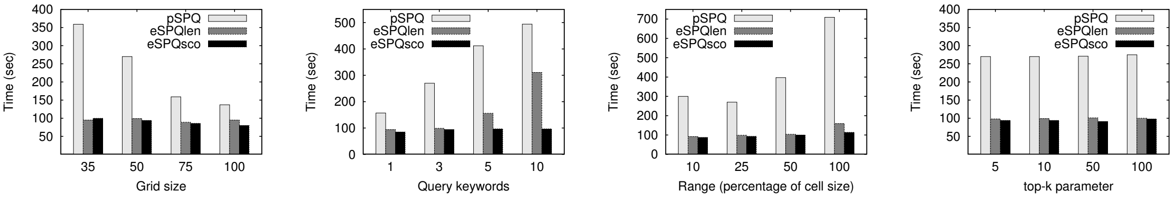
Query generation. Queries are generated by selecting various values for the query radius r and a number of random query keywords $q.\mathcal{W}$ from the vocabulary of the respective dataset[2].

[2] We also explored alternative methods for keyword selection instead of random selection, such as selecting from the most frequent words or the least frequent words, but the execution time of our algorithms was not significantly affected.



(a) Varying grid size. (b) Varying number of query keywords. (c) Varying query radius. (d) Varying k.

Figure 5: Experiments for Flickr (FL) dataset.



(a) Varying grid size. (b) Varying number of query keywords. (c) Varying query radius. (d) Varying k.

Figure 6: Experiments for Twitter (TW) dataset.

Parameters. During the experimental evaluation a number of parameters were varied in order to observe their effect on each algorithm's runtime. These parameters, reported in Table 3, are: (i) the radius of the query, (ii) the number of keywords of the query, (iii) the size of the grid that we use to partition the data space, (iv) the number of the k results that the algorithm returns, and (v) the size of the dataset. In all cases, the number of Reducers is set equal to the number of cells in the spatial grid.

Metrics. The algorithms are evaluated by the time required for the MapReduce job to complete, i.e., the job execution time.

7.2 Experimental Results

7.2.1 Experiments with Real Data: Flickr

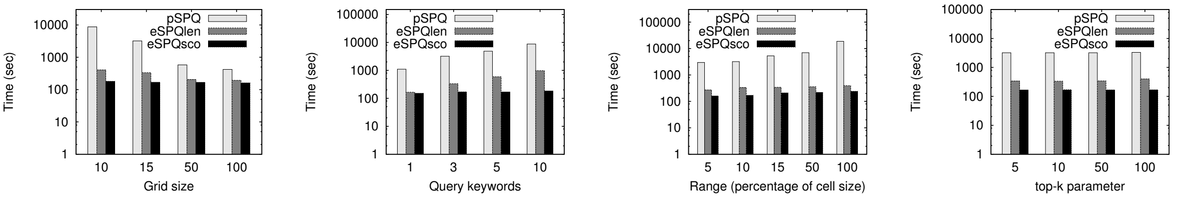
Figure 5 presents the results obtained for the Flickr (FL) dataset. First, in Figure 5(a), we study the effect of grid size to the performance of our algorithms. The first observation is that using more grid cells (i.e., Reduce tasks) improves the performance, since more, yet smaller, parts of the problem need to be computed. The algorithms that employ early termination (*eSPQlen*, *eSPQsco*) are consistently much faster than the grid-based algorithm *pSPQ*. In particular, *eSPQsco* improves *pSPQ* up to a factor of 6x. Between the early termination algorithms, *eSPQsco* is consistently faster due to the sorting based on score, which typically needs to access only a handful of feature objects before reporting the correct result. Figure 5(b) shows the effect of varying the number of query keywords ($|q.\mathcal{W}|$). In general, when more keywords are used in the query more feature objects are passed to the Reduce phase, since the probability of having non-zero Jaccard similarity increases. This is more evident in *pSPQ*, whose cost increases substantially with increased query keyword length. Instead, *eSPQsco* is not significantly affected by the increased number of keywords, because it still manages to find the correct result after examining only few feature objects. This experiment demonstrates the value of the early termination criterion employed in *eSPQsco*. In Figure 5(c), we gradually increase the radius of the query. In principle, this makes query processing more costly as again more feature objects become candidates for determining the score of any data object. However, the early termination algorithms are not significantly affected by increased values of radius, as they can still report the correct result after examining few feature objects only. Finally, in Figure 5(d), we study the effect on increased values of top-k. The chart shows that all algorithms are not particularly sensitive to increased values of k, because the cost of reporting a few more results is marginal compared to the work needed to report the first result.

7.2.2 Experiments with Real Data: Twitter

Figure 6 depicts the results obtained in the case of the Twitter (TW) dataset. In general, the conclusions drawn are quite similar to the case of the FL dataset. The algorithms that employ early termination, and in particular *eSPQsco*, scale gracefully in all setups. Even in the harder setups of many query keywords (Figure 6(b)) and larger query radius (Figure 6(c)), the performance of *eSPQsco* is not significantly affected. This is because as soon as the first few feature objects with highest scores are examined, the algorithm can safely report the top-k data objects in the cell. In other words, the vast majority of feature objects assigned to a cell are not actually processed, and exactly this characteristic makes the algorithm scalable both in the case of more demanding queries as well as in the case of larger datasets.

7.2.3 Experiments with Uniform Data

In order to study the performance of our algorithms for large-scale datasets, we employ in our study synthetic datasets. Figure 7 presents the results obtained for the Uniform (UN) dataset. Notice the log-scale used in the y-axis. A general observation is that *eSPQsco* that uses early termination out-



(a) Varying grid size. (b) Varying number of query keywords. (c) Varying query radius. (d) Varying k.

Figure 7: Experiments for Uniform (UN) dataset.

performs *pSPQ* by more than one order of magnitude. This is a strong indication in favor of the algorithms employing early termination, as their performance gains are more evident for larger datasets, such as the synthetic ones. Also, the general trends are in accordance with the conclusions derived from the real datasets. It is noteworthy that the performance of *eSPQsco* remains relatively stable in the harder setups consisting of many query keywords (Figure 7(b)) and larger query radius (Figure 7(c)).

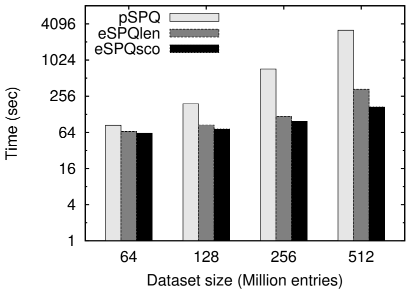


Figure 8: Scalability of all algorithms.

Moreover, Figure 8 shows the results obtained when we vary the dataset size. This experiment aims at demonstrating the nice scaling properties of our algorithms. In particular, *pSPQ* scales linearly with increased dataset size, which is already a good result. However, the algorithms that employ early termination perform much better, since they only examine few feature objects regardless of the increase of dataset size. The experiment also shows that the gain of the algorithms that employ early termination compared to *pSPQ* increases for larger datasets.

7.2.4 Experiments with Clustered Data

Figure 9 presents the results obtained for the Clustered (CL) dataset. It should be noted that such a data distribution is particularly challenging as: (a) it is hard to fairly assign the objects to Reducers, thus typically some Reducers are overburdened, and (b) excessive object duplication can occur when a cluster is located on grid cell boundaries. For the CL dataset, we observed that *pSPQ* results in extremely high execution time, thus it is not depicted in the charts. For instance, for the default setup, it takes approximately 48 hours for *pSPQ* to complete. This is due to the fact that some Reducers are assigned with too many feature objects and *pSPQ* has to perform $O(|O_i| \cdot |F_i|)$ score computations before termination. Still, the algorithms employing early termination perform much better in all cases. Again, when *eSPQsco* is considered, its performance is the best among all algorithms, and it remains quite stable even in the case of more demanding queries. This experiment verifies the nice properties of *eSPQsco*, even for the combination of large-scale dataset with a demanding data distribution.

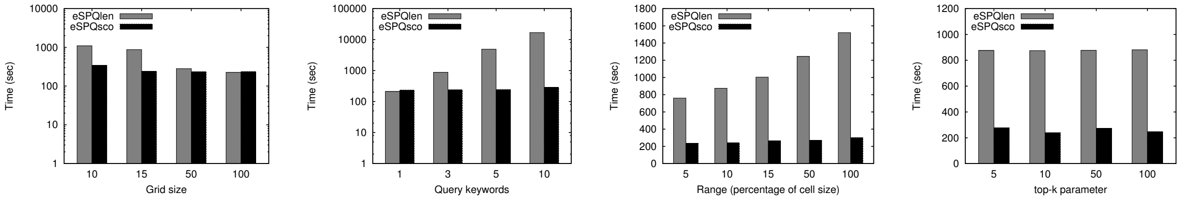
8. RELATED WORK

In this section, we provide a brief overview of related research efforts.

Spatial Preference Queries. The spatial preference query has been originally proposed in [16] and later extended in [17]. Essentially, this query enables the retrieval of interesting spatial locations based on the quality of other facilities located in their vicinity. Rocha et al. [12] propose efficient query processing algorithms based on a mapping in score-distance space that enables the materialization of sufficient pairs of data and feature objects. Ranking of data objects based on their spatial neighborhood without supporting keywords has also been studied in [7, 15]. As already mentioned, none of these approaches support keyword-based retrieval.

Spatio-textual Queries. Object retrieval based on a combination of spatial and textual information is a highly active research area recently. We refer to [3] for an interesting overview of query types along with an experimental comparison. The query studied in this paper has similarities to spatio-textual joins. Spatio-textual similarity join in a centralized setting is studied in [1], while a ranked version of this join which does not require thresholds from the user is studied in [10]. Partitioning strategies that also support multi-threaded processing of spatio-textual joins are examined in [11]. The spatial group keyword query [2] retrieves a group of objects located near the query location, such that the union of their textual descriptions covers the query keywords. In [5], the best keyword cover query is introduced, which retrieves a set of spatial objects that together cover the query keywords and additionally are located nearby. The most relevant query to our work is the ranked spatio-textual preference query proposed in [14]; in this paper, we study a distributed variant of this query. Nevertheless, all the above works target centralized environments, and their adaptation to distributed, large-scale settings is not straightforward.

Spatio-textual Queries at Scale. The current trend for scalable query processing is to employ a parallel processing solution based on MapReduce. For a survey on query processing in MapReduce we refer to [6]. To the best of our knowledge, the area of spatio-textual query processing at scale is not explored yet. Existing systems for parallel and scalable data processing in the context of MapReduce include SpatialHadoop [8]. However, SpatialHadoop targets



(a) Varying grid size. (b) Varying number of query keywords. (c) Varying query radius. (d) Varying k.

Figure 9: Experiments for Clustered (CL) dataset.

spatial data and is not optimized for spatial data annotated with textual descriptions. Spatial joins (binary and multi-way joins) in a MapReduce context are also studied in [18, 9] respectively, but again no provision for textual annotations exists. In [19], an approach for spatio-textual similarity join in MapReduce is presented, where pairs of spatio-textual objects located within a user-specified distance and having textual similarity over a user-specified threshold are retrieved. However, the query in [19] targets a single dataset (essentially a self-join) without ranking and without keywords as user input. Another substantial difference to our work is that their solution requires multiple MapReduce phases (jobs), whereas our algorithms consist of a single MapReduce job. Finally, a recent work on ranked query processing (top-k joins) in MapReduce is presented in [13], which employs early termination in a different way (in the Map phase) from this work (in the Reduce phase).

9. CONCLUSIONS

In this paper we study the problem of parallel/distributed processing of spatial preference queries using keywords. We propose scalable algorithms that rely on grid-based re-partitioning of input data in order to generate partitions that can be processed independently in parallel. To boost the performance of query processing, we employ early termination, thus reporting the correct result set after examining only a handful of the input data points. Our experimental study shows that our best algorithm consistently outperforms the remaining ones, and its performance is not significantly affected even in the case of demanding queries.

Acknowledgments

The work of C. Doulkeridis, A. Vlachou, and D. Mpestas 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. The work of N. Mamoulis was supported by EU-funded Marie Curie Reintegration Grant project titled LBSKQ.

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, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *Proc. of SIGMOD*, pages 373–384, 2011.

[3] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*, 6(3):217–228, 2013.

[4] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of OSDI*, 2004.

[5] K. Deng, X. Li, J. Lu, and X. Zhou. Best keyword cover search. *IEEE Trans. Knowl. Data Eng.*, 27(1):61–73, 2015.

[6] C. Doulkeridis and K. Nørvåg. A survey of analytical query processing in MapReduce. *VLDB Journal*, 2014.

[7] Y. Du, D. Zhang, and T. Xia. The optimal-location query. In *Proc. of SSTD*, pages 163–180, 2005.

[8] A. Eldawy and M. F. Mokbel. SpatialHadoop: A MapReduce framework for spatial data. In *Proc. of ICDE*, pages 1352–1363, 2015.

[9] H. Gupta, B. Chawda, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. K. Mohania. Processing multi-way spatial joins on MapReduce. In *Proc. of EDBT*, pages 113–124, 2013.

[10] H. Hu, G. Li, Z. Bao, J. Feng, Y. Wu, Z. Gong, and Y. Xu. Top-k spatio-textual similarity join. *IEEE Trans. Knowl. Data Eng.*, 28(2):551–565, 2016.

[11] J. Rao, J. J. Lin, and H. Samet. Partitioning strategies for spatio-textual similarity join. In *Proc. of BigSpatial Workshop*, pages 40–49, 2014.

[12] 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.

[13] M. Saouk, C. Doulkeridis, A. Vlachou, and K. Nørvåg. Efficient processing of top-k joins in MapReduce. In *Proc. of IEEE Big Data*, 2016.

[14] G. Tsatsanifos and A. Vlachou. On processing top-k spatio-textual preference queries. In *Proc. of EDBT*, pages 433–444, 2015.

[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 Trans. Knowl. Data Eng.*, 23(3):433–446, 2011.

[18] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu. SJMR: parallelizing spatial join with MapReduce on clusters. In *Proc. of CLUSTER*, pages 1–8, 2009.

[19] Y. Zhang, Y. Ma, and X. Meng. Efficient spatio-textual similarity join using MapReduce. In *Proc. of IEEE Web Intelligence*, pages 52–59, 2014.

Continuous Imputation of Missing Values in Streams of Pattern-Determining Time Series

Kevin Wellenzohn   Michael H. Böhlen
Department of Computer Science
University of Zurich, Switzerland
{wellenzohn, boehlen}@ifi.uzh.ch

Anton Dignös   Johann Gamper
Hannes Mitterer
Faculty of Computer Science
Free University of Bolzano, Italy
firstname.lastname@unibz.it

ABSTRACT

Time series data is ubiquitous but often incomplete, e.g., due to sensor failures and transmission errors. Since many applications require complete data, missing values must be imputed before further data processing is possible.

We propose Top-k Case Matching (TKCM) to impute missing values in streams of time series data. TKCM defines for each time series a set of reference time series and exploits similar historical situations in the reference time series for the imputation. A situation is characterized by the *anchor point* of a pattern that consists of l consecutive measurements over the reference time series. A missing value in a time series s is derived from the values of s at the anchor points of the k most similar patterns. We show that TKCM imputes missing values *consistently* if the reference time series *pattern-determine* time series s, i.e., the pattern of length l at time t_n is repeated at least k times in the reference time series and the corresponding values of s at the anchor time points are similar to each other. In contrast to previous work, we support time series that are not linearly correlated but, e.g., phase shifted. TKCM is resilient to consecutively missing values, and the accuracy of the imputed values does not decrease if blocks of values are missing. The results of an exhaustive experimental evaluation using real-world and synthetic data shows that we outperform the state-of-the-art solutions.

1. INTRODUCTION

Time series data appears in many application domains, e.g., meteorology, sensor networks, the financial world, and network monitoring. Often time series data is incomplete with values missing because of sensor failures, transmission errors, etc. Many applications require complete data, hence missing values must be recovered before further data processing is possible.

Our research is motivated by the problem of missing values in the data collected by the *Südtiroler Beratungsring für Obst- und Weinbau* (SBR). The SBR monitors and analyzes meteorological data streams in real time and alerts wine and apple farmers of potential harvest threats, such as frost, apple scab, and fire blight. The SBR operates a network of more than 130 weather stations in South Tyrol, each of which records approximately 20 meteorological parameters at a sample rate of five minutes. The measurements date back to 2007 with a total of 88.9M measurements. For illustration purposes, we use the temperature taken at one meter above ground level, which ranges from -20.3°C to $+40.3$°C and has a total of 7.8M ($= 8\%$) missing values. Currently, missing values are manually imputed by domain experts, based on the values at neighboring stations.

Various works have observed that time series are correlated and imputation techniques have been proposed that exploit the information of co-evolving time series [12, 13, 14, 16, 25]. Popular solutions include SVD based matrix decomposition techniques [11, 12], multivariate autoregression analysis [25], and PCA (principal component analysis) guided data summarization [16, 17, 23]. These approaches perform well if the time series are linearly correlated according to the Pearson correlation. The imputation accuracy deteriorates if the time series are shifted and have a Pearson correlation close to zero.

In this paper, we propose Top-k Case Matching (TKCM) to impute missing values in streams of non-linearly correlated time series. TKCM defines for each time series s a small set $\mathbf{R}_s$ of *reference time series*. If the value in s at the current time t_n is missing, TKCM defines a *query pattern* $P(t_n)$ that is *anchored* at t_n and composed of the l most recent measurements of the reference time series. Then, the k most similar non-overlapping patterns to the query pattern within a given time window are determined. The missing value is derived from the values of time series s at the anchor points of the k most similar patterns. This process is illustrated in Fig. 1, where the value of the time series s at the current time t_n is missing (small circle on the right). There are two reference time series of s, i.e., $\mathbf{R}_s = \{r_1, r_2\}$. The query pattern $P(t_n)$ is composed of the snippets of the reference time series in the black frame. The $k = 2$ most similar patterns to the query pattern are anchored at t_i and t_j and are shown as dashed rectangles. The missing value of s is derived from the values of s at the anchor points t_i and t_j (small circles).

TKCM exploits two common properties of time series. First, time series often exhibit (not necessarily regularly) repeating patterns, also referred to as seasonal patterns. Second, time series are (not necessarily linearly) correlated in the sense that, whenever a pattern in a set of reference time series repeats, time series s exhibits similar values. If these two properties are satisfied we say that at time t_n the reference time series $\mathbf{R}_s$ *pattern-determine* time series s, denoted by $\mathbf{R}_s, t_n \xrightarrow{\text{pd}} s$. In other words, whenever similar patterns occur in $\mathbf{R}_s$, the values of time series s are similar to each other, too. In contrast to previous work, this property allows not only linearly correlated reference time series but permits phase shifts. For instance, in Fig. 1 the two (shifted) reference time se-

©2017, Copyright is with the authors. Published in Proc. 20th International Conference on Extending Database Technology (EDBT), March 21-24, 2017 - Venice, Italy: ISBN 978-3-89318-073-8, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

Series ISSN: 2367-2005

10.5441/002/edbt.2017.30

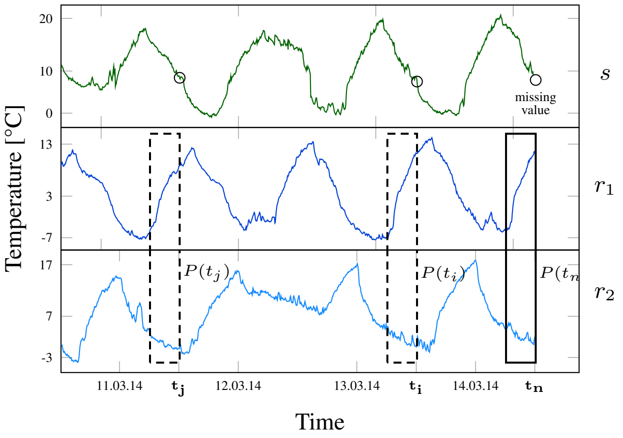


Figure 1: Imputation of a missing value of s at time t_n.

ries $\mathbf{R}_s = \{r_1, r_2\}$ *pattern-determine* time series s at time t_n. We show that TKCM imputes missing values consistently if the time series are pattern determining.

The paper makes the following technical contributions:

- We present and formalize Top-k Case Matching (TKCM) to impute missing values in streams of pattern-determining time series, which covers non-linear relationships between time series.
- We show that TKCM computes correct results for shifted time series that are not linearly correlated. We use a pattern of length $l > 1$ to exploit several consecutive measurements to find similar historical situations.
- We propose a dynamic programming scheme to find the k non-overlapping patterns that minimize the sum of dissimilarities with respect to the query pattern.
- We empirically show on real-world and synthetic datasets that TKCM: (a) outperforms state-of-the-art solutions, (b) can impute values in time series with phase shifts, and (c) is resilient to large blocks of consecutively missing values.

The paper is structured as follows. Section 2 discusses related works. After the preliminaries in Section 3 we describe our approach in Section 4. Section 5 analyzes the properties of TKCM and works out the differences between linear and non-linear correlations. In Section 6 we provide an implementation of TKCM that uses a dynamic programming solution to find the k non-overlapping patterns that are most similar to the query pattern. We continue with the experimental evaluation in Section 7 before we conclude this paper and present future research directions in Section 8.

2. RELATED WORK

The need to recover missing data arises in many applications, ranging from meteorology [18, 26], to social science [20], machine learning [2], motion capture systems [14] and DNA microarray analysis [24]. Imputing a missing value means to recover it with a good estimate that is derived from intrinsic relationships in the underlying dataset.

Simple imputation techniques include *mean* and *mode* imputation [2], which replace the missing value with the mean or mode of the same attribute. Interpolation techniques, such as linear interpolation and spline interpolation, estimate the missing value from immediately preceding and succeeding values of the same attribute. If the gap is long, i.e., if many consecutive values are missing, these interpolation techniques perform badly. For instance, if an entire period of a sine wave is missing, linear interpolation would replace the gap with a straight line. Regression methods [25] estimate the missing value of a time series (e.g., temperature in Zurich) based on the value of other time series (e.g., temperature in Bern and Basel). Paulhus et al. [18] observed that nearby weather stations have similar values and computed a missing value at one station as the average of the values of nearby stations. Yozgatligil et al. [26] give a recent survey of imputation methods for meteorological time series and cover approaches based on neural networks and multiple imputation [19].

The ARIMA model [4] is a popular time series forecasting model that is a generalization of the auto-regressive (AR) model. ARIMA assumes a linear dependency of unknown future values on known past values of the time series. Finding the proper values for p, d, q in an ARIMA(p, d, q) model is tedious, complex and involves manual analysis, known as the Box-Jenkins methodology [4].

Batista et al. [2] study the problem of missing data in the context of machine learning algorithms and present the k-Nearest Neighbor Imputation (kNNI) method to recover these values. For a multi-attribute object (e.g., breast cancer test with multiple measurements) that has a missing value for one attribute A, the kNNI approach looks for k objects with similar values for the other attributes according to a distance metric that is not specified. The missing value is derived from the values of attribute A in these objects. Troyanskaya et al. [24] extend kNNI to weight the k most similar items according to their similarity. Our approach, TKCM, uses the concept of nearest neighbors (k most similar patterns), but is designed for time series streams and uses a two-dimensional query pattern for which the k most similar *non-overlapping* patterns according to the Euclidean distance are searched.

Khayati et al. [11] propose REBOM to recover blocks of missing values in irregular time series with non-repeating trends. The algorithm builds a matrix which stores the incomplete time series and the n most linearly correlated time series according to Pearson correlation. Missing values are first initialized, e.g., using linear interpolation. Then the matrix is iteratively decomposed using the *Singular Value Decomposition* (SVD) method, where the least significant singular values are truncated. Due to the quadratic runtime complexity, REBOM does not scale to long time series. Next, Khayati et al. [12] present a solution with linear space complexity based on the *Centroid Decomposition* (CD), which is an approximation of SVD. Unlike our approach, SVD and CD assume a linear correlation between an incomplete time series and its reference time series. If time series are not linearly correlated, the imputation accuracy deteriorates since these trends are captured by the truncated least significant singular values. Khayati et al. [13] show that CD imputes more accurately than SVD when some reference time series are shifted and hence lowly linearly-correlated, because CD prioritizes highly linearly-correlated reference time series. Nevertheless, their experiments show that adding more lowly-correlated reference time series has a negative impact on CD's accuracy.

Sorjamaa et al. [15, 22] propose an imputation method based on a Self-Organizing-Map (SOM), which is an unsupervised learning technique based on neural networks. A combination of SVD and SOM [22] uses SVD for the imputation after initializing missing values in the matrix by a SOM classifier, whereas [15] combines two SOM classifiers for the imputation. Both methods are only evaluated on linearly correlated time series.

DynaMMo [14] is used for mining, summarizing, and imputing time series extracted from human motion capture systems. It is based on Kalman filters, which, similar to SVD, assume a linear correlation between time series to accurately estimate unknown values. Moreover, unlike our approach, DynaMMo allows only one reference time series, which often is insufficient for an accurate imputation.

The works most similar to our approach are MUSCLES [25] and SPIRIT [16, 17, 23], which focus on the online imputation of missing values in streams of time series data. Both algorithms use variants of auto-regressive (AR) models and exploit linear correlations between data streams. When the linear correlation diminishes, as in the case of shifted time series, none of the two approaches performs well.

MUSCLES [25] is an online algorithm that is based on a *multivariate* auto-regression model, whose parameters are incrementally updated using the Recursive Least Squares method. Besides past values of the incomplete time series, MUSCLES takes also the most recent values of co-evolving and linearly correlated time series into account that are within a window p. How to choose p is not discussed; in the experiments $p = 6$ is used. After p consecutive missing values, MUSCLES relies exclusively on imputed values for the incomplete time series. Since small imputation inaccuracies accumulate over a long stretch of missing values, MUSCLES accuracy deteriorates. Additionally, MUSCLES does not scale well to a large number of streams, unless an expensive offline subset selection on the time series is performed [17].

SPIRIT [16, 17, 23] uses an online Principal Component Analysis (PCA) to reduce a set of n co-evolving and correlated streams to a small number of k hidden variables that summarize the most important trends in the original data. For each hidden variable, SPIRIT fits one AR model on past values, which is incrementally updated as new data arrives. If a value is missing, the AR models are used to forecast the current value of each variable, from which an estimate of the missing value is derived. The imputed value, along with the non-missing values, is then used to update the forecasting models. Updating the models with imputed models incurs similar problems as MUSCLES since inaccuracies are propagated. Since PCA and SVD are based on the same underlying principle, PCA shares SVD's weaknesses for shifted time series.

From an implementation perspective, TKCM needs to find similar patterns in time series. This problem has been studied extensively for a single time series, yielding different dimensionality reduction techniques, e.g., Discrete Fourier Transform [6], Piecewise Aggregate Approximation [7], and iSAX [21]. Keogh et al. [10] present a fast approach to find a subsequence (i.e., one-dimensional pattern), termed shapelet, of a time series that is most representative for a set of time series. Finding patterns in our approach is more complex. We seek patterns that span several time series, and we have to select the k most similar non-overlapping patterns. The main focus of this work is not performance, but an accurate imputation of shifted time series streams.

3. PRELIMINARIES

Consider a set $\mathbf{S} = \{s_1, s_2, \ldots\}$ of streaming time series. Each time series reports values from a sensor measured at time points $\ldots, t_{n-2}, t_{n-1}, t_n$, where t_n denotes the current time, i.e., the time of the latest measurement. The value of a time series $s \in \mathbf{S}$ at time t_i is denoted as $s(t_i)$. We write $s(t_n) = \textsc{Nil}$ to denote that the current value of s is missing. $W = \{t_{n-L+1}, \ldots, t_{n-1}, t_n\}$ denotes the L time points in our streaming window for which we keep measurements in main memory. We assume that the streaming window W is long enough to include the query pattern and k non-overlapping similar patterns.

For each time series $s \in \mathbf{S}$ there exists an ordered sequence $\langle r_1, r_2, \ldots \rangle$ of candidate reference time series, where $r_i \in \mathbf{S} \setminus \{s\}$. They have been identified by domain experts and are consulted if the current value in s is missing and must be recovered. The candidate reference time series of s are ranked according to how suitable they are for imputing a missing value in s. A single reference time series does not yield a robust method to estimate a missing value. Instead the d best candidate reference time series that do not have a missing value at the current time t_n are used. Let $s \in \mathbf{S}$ be an incomplete time series with $s(t_n) = \textsc{Nil}$. The *reference time series* $\mathbf{R}_s$ for s at the current time t_n are the first d time series in the ordered sequence for which $r(t_n) \neq \textsc{Nil}$.

Note that there can be multiple incomplete time series with a missing value at t_n. For each incomplete time series s_i its missing value $s_i(t_n)$ is imputed individually using the respective set of reference time series $\mathbf{R}_{s_i}$.

Example 1. As a running example, we use the four time series in Table 2. The current time is t_n = 14:20. Time series s is incomplete, hence the missing value at 14:20 must be imputed. We assume a sliding window of one hour, containing $L = 12$ measurements. For all time points before t_n the values either have been reported by the sensor or have been imputed, e.g., r_2(13:40) $= \widehat{18.8}$°C. The candidate reference time series are $\langle r_1, r_2, r_3 \rangle$. At the current time t_n = 14:20, the $d = 2$ reference time series for s are $\mathbf{R}_s = \{r_1, r_2\}$. When the current time was t_n = 13:40, we had $\mathbf{R}_s = \{r_1, r_3\}$ since r_2(13:40) was missing. □

Notation	**Description**
t_n	Current time
$\mathbf{S} = \{s_1, s_2, \ldots\}$	Set of time series
$s(t_n) = \textsc{Nil}$	Missing value of time series s at time t_n
$\hat{s}(t_n) \neq \textsc{Nil}$	Imputed value of time series s at time t_n
d	Number of reference time series
$\mathbf{R}_s = \{r_1, \ldots, r_d\}$	Set of d reference time series for s
$W = \{\ldots, t_n\}$	Time points in streaming window
L	Length of streaming window W
l	Pattern length
$P(t_i)$	Pattern anchored at time t_i
k	Number of anchor points
$\mathbf{A} = \{t_{i_1}, \ldots, t_{i_k}\}$	k most similar anchor points

Table 1: Summary of notation.

4. TOP-K CASE MATCHING (TKCM)

4.1 Approach

For the recovery of a missing value in an incomplete time series we look for patterns in the past when the values of the reference time series were similar to the current values.

Definition 1. (*Pattern*) Let $\mathbf{R}_s = \{r_1, \ldots, r_d\}$ be the reference time series for an incomplete time series s. The *pattern* $P(t_i)$ of length $l > 0$ over $\mathbf{R}_s$ that is anchored at time t_i is defined as a $d \times l$ matrix $P(t_i)$ as follows:

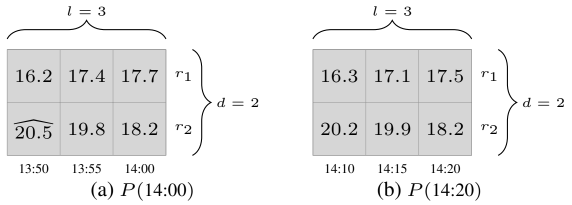
$$\begin{aligned} P(t_i) = (&(r_1(t_{i-l+1}), \ldots, r_1(t_i)),\\ &\quad\vdots \qquad\qquad\quad \vdots\\ &(r_d(t_{i-l+1}), \ldots, r_d(t_i))). \end{aligned}$$

Time t	$\cdots$	13:25	13:30	13:35	13:40	13:45	13:50	13:55	14:00	14:05	14:10	14:15	14:20
s	$\cdots$	22.8°C	21.4°C	21.8°C	$\widehat{23.1}$°C	23.5°C	22.8°C	21.2°C	21.9°C	23.5°C	22.8°C	21.2°C	NIL
r_1	$\cdots$	16.5°C	17.2°C	17.8°C	16.6°C	15.8°C	16.2°C	17.4°C	17.7°C	15.3°C	16.3°C	17.1°C	17.5°C
r_2	$\cdots$	20.3°C	19.8°C	18.6°C	$\widehat{18.8}$°C	$\widehat{20.0}$°C	$\widehat{20.5}$°C	19.8°C	18.2°C	20.1°C	20.2°C	19.9°C	18.2°C
r_3	$\cdots$	14.0°C	14.8°C	13.6°C	13.0°C	14.5°C	14.3°C	14.0°C	15.0°C	13.0°C	14.5°C	14.3°C	14.6°C

Table 2: Time series s with a missing value at time t_n = 14:20 and the three reference time series r_1, r_2 and r_3.

A pattern is anchored at a time point t_i and consists of the values from t_{i-l+1} to t_i of each reference time series. Each row represents a subsequence of a reference time series, and each column represents the values of the reference time series at a time point. The pattern contains for each reference time series only the values at time t_i, if $l = 1$. The pattern includes additionally the preceding $l - 1$ values, and hence captures the trend, if $l > 1$.

Example 2. Figure 2 shows two patterns over the reference time series $\mathbf{R}_s = \{r_1, r_2\}$ in our running example (cf. Table 2). Both patterns have length $l = 3$ and are anchored at time points 14:00 and 14:20, respectively. Pattern $P(14{:}00)$ contains one imputed value, namely $r_2(13{:}50) = \widehat{20.5}$°C. Since $l > 1$ the pattern captures the current trend of the time series. □



(a) $P(14{:}00)$ (b) $P(14{:}20)$

Figure 2: Two patterns of length $l = 3$ over $d = 2$ reference time series.

The pattern that is anchored at the current time is termed *query pattern* $P(t_n)$. We search in the reference time series for the k patterns that are most similar to $P(t_n)$ using the L_2 norm.

Definition 2. (*Pattern Dissimilarity*) Let s be an incomplete time series with reference time series $\mathbf{R}_s$ at time t_n. The *dissimilarity*, δ, between two patterns $P(t_m)$ and $P(t_n)$ is defined as

$$\delta(P(t_m), P(t_n)) = \sqrt{\sum_{r_i \in \mathbf{R}_s} \sum_{0 \leq j < l} (r_i(t_{m-j}) - r_i(t_{n-j}))^2}.$$

Example 3. The dissimilarity between the two patterns in Figure 2 is computed as follows: $\delta(P(14{:}00), P(14{:}20)) = \sqrt{(17.7 - 17.5)^2 + (17.4 - 17.1)^2 + (16.2 - 16.3)^2 + \ldots} = 0.43$. □

The dissimilarity measure is used to determine the k most similar patterns to query pattern $P(t_n)$. The anchor time points of these k patterns are referred to as the k most similar anchor points $\mathbf{A}$.

Definition 3. (*k Most Similar Anchor Points*) Let $P(t_n)$ be the query pattern for incomplete time series s at time t_n with reference time series $\mathbf{R}_s$, and L be the length of the streaming time window. The k most similar *anchor points* to t_n are a set $\mathbf{A} \subseteq W$, with $|\mathbf{A}| = k$, for which the following holds:

$$\forall t \in \mathbf{A} : t_{n-L+l} \leq t \leq t_{n-l} \tag{1}$$

$$\forall t, t' \in \mathbf{A} : t \neq t' \rightarrow |t - t'| \geq l \tag{2}$$

$$\forall \mathbf{A}' : (1) \wedge (2) \wedge |\mathbf{A}'| = k \rightarrow \sum_{t_i \in \mathbf{A}} \delta(P(t_i), P(t_n)) \leq \sum_{t_i \in \mathbf{A}'} \delta(P(t_i), P(t_n)) \tag{3}$$

The first condition states that all patterns are within the time window and do not overlap $P(t_n)$. The second condition states that the patterns do not overlap each other. The third condition ensures that the patterns that are anchored at the time points in $\mathbf{A}$ minimize the sum of the dissimilarities with respect to query pattern $P(t_n)$.

We pick only *non-overlapping* patterns to avoid near duplicates [5, 8]. Our experiments have shown that if overlapping patterns were allowed, the k most similar anchor points for some pattern $P(t_i)$ are frequently time points t_{i+1} and t_{i-1}, which anchor the first and second most similar patterns, etc. This is clearly not desired. Instead, non-overlapping patterns guarantee that we find a diverse set of patterns on which the imputation is based.

The missing value in the incomplete time series s is the average of the values of s at the most similar time points.

Definition 4. (*Imputed Value*) Let s be a time series with reference time series $\mathbf{R}_s$ at t_n and missing value $s(t_n)$. Furthermore, let $\mathbf{A}$ be the k most similar time points to the current time. The *imputed value* $\hat{s}(t_n)$ for time series s at time t_n is

$$\hat{s}(t_n) = \frac{1}{k} \sum_{t \in \mathbf{A}} s(t). \tag{4}$$

Example 4. Figure 3 shows a graphical representation of our running example (cf. Table 2). The value of s at time t_n = 14:20 is missing and must be imputed. The query pattern $P(14{:}20)$ is framed in black. The two patterns most similar to the query pattern are shown as dashed rectangles and are anchored at time 14:00 and 13:35, respectively. Thus, $\mathbf{A} = \{14{:}00, 13{:}35\}$ are the anchor points. The missing value is computed as the average of the values $s(14{:}00)$ and $s(13{:}35)$: $\hat{s}(14{:}20) = (21.9°\text{C} + 21.8°\text{C})/2 = 21.85°\text{C}$. □

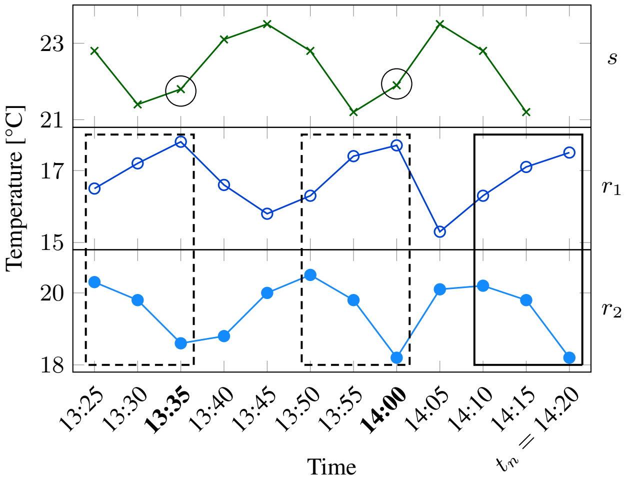


Figure 3: The $k = 2$ most similar non-overlapping patterns for query pattern $P(14{:}20)$ are $P(14{:}00)$ and $P(13{:}35)$.

5. ANALYSIS

5.1 Correlation

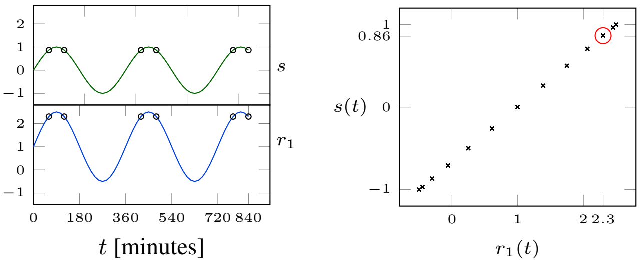
A salient property of TKCM is its ability to handle time series that are shifted and hence not linearly correlated. The Pearson correlation, the most common correlation measure, quantifies the degree of linear correlation between time series s and r, as

$$\rho(s, r) = \frac{\sum_{t \in W} (s(t) - \bar{s})(r(t) - \bar{r})}{\sqrt{\sum_{t \in W} (s(t) - \bar{s})^2} \sqrt{\sum_{t \in W} (r(t) - \bar{r})^2}},$$

where $\bar{s}$ and $\bar{r}$ are the means of, respectively, s and r in window W. Pearson correlation ranges from -1 to 1, indicating total negative and positive correlation, respectively. Thus, s and r are linearly correlated if $|\rho(s, r)|$ is high. If $\rho(s, r) = 0$ time series s and r are not linearly correlated.

Intuitively, a linear correlation ensures that (a) if one time series has close values for two time points, also the other has close values for these two time points and (b) if one time series has far apart values for two time points, also the other has far apart values for these two time points.

Example 5. Consider Figure 4a with time series $s(t) = \text{sind}(t)$ and $r_1(t) = 1.5 \times \text{sind}(t) + 1$, having different amplitudes and offsets. The value of r_1 at $t = 840$ is $r_1(840) = 2.3$, and the same value $r_1(t)$ appears for time points $t \in \{780, 480, 420, 120, 60\}$. Figure 4a illustrates that these are exactly the time points for which s has the same value of $s(840) = 0.86$. Thus, the time series are perfectly linearly correlated. Figure 4b uses a scatterplot to display the correlation between s and r_1. The scatterplot displays for each time point t the point $(r_1(t), s(t))$. For instance, at time $t = 840$ we have $r_1(840) = 2.3$ and $s(840) = 0.86$ and the point $(2.3, 0.86)$ is displayed in the scatterplot. The more the scatterplot resembles a line with a non-zero slope, the higher the linear correlation and hence Pearson correlation. □



(a) Time series s and r_1

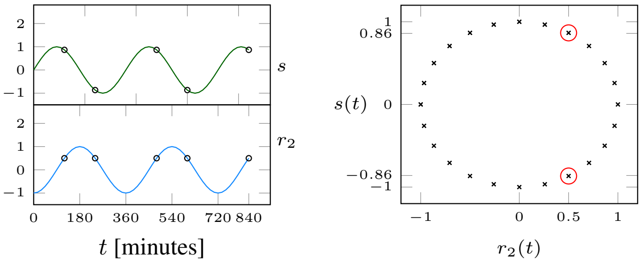
(b) Scatterplot of linear correlation

Figure 4: Linearly correlated time series $s(t) = \text{sind}(t)$ and $r_1(t) = 1.5 \times \text{sind}(t) + 1$.

Example 5 illustrates how the imputation for linearly correlated time series works. If $s(t_n)$ is missing, we know that whenever time series r_1 observes value $r_1(t_n)$ (e.g., 2.3), time series s observes the same value $s(t)$ (e.g., 0.86). Hence we can use value $s(t)$ for any t where $r_1(t) = 2.3$ to impute value $s(t_n)$.

In contrast, if the Pearson correlation approaches zero, s can have very different values although the reference time series has the same value. This is illustrated in Example 6.

Example 6. Figure 5a depicts the time series $s(t) = \text{sind}(t)$ and $r_2(t) = \text{sind}(t - 90)$. The two time series have the same amplitude and offset but they are phase shifted. Time series r_2 has the value $r_2(840) = 0.5$ also for time points $t \in \{600, 480, 240, 120\}$. However, s has different values, i.e., value $s(t) = 0.86$ for time points $t \in \{480, 120\}$ and value $s(t) = -0.86$ for time points $t = \{600, 240\}$. The scatterplot in Figure 5b shows that the data points do not cluster around a line, which means that they are non-linearly correlated. Their Pearson correlation is -0.0085. Note that for the same value of $r_2(t)$ we can have two different values for $s(t)$. For instance, for $r_2(t) = 0.5$ we have either $s(t) = 0.86$ or $s(t) = -0.86$. □



(a) Time series s and r_2

(b) Scatterplot of non-linear correlation

Figure 5: Non-linearly correlated time series $s(t) = \text{sind}(t)$ and $r_2(t) = \text{sind}(t - 90)$

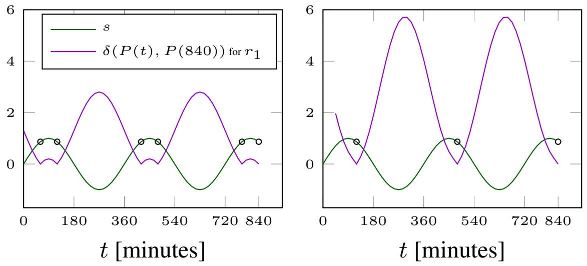
Example 6 illustrates the key problem with non-linear correlations: the values of one time series can no longer be used to reliably determine a missing value in another time series. In the experiments we will see that this leads to imputations with a high root mean square error.

5.2 Pattern Length

The previous section illustrated that shifted time series that are not linearly correlated are difficult to handle. Intuitively, for shifted time series it is not sufficient to consider a single point in time. Instead it is necessary to consider a pattern that includes neighboring time points to correctly relate time series. This section illustrates that a pattern with a length $l > 1$ improves the imputation for non-linear correlations. We write $P_l(t)$ to denote a pattern of length l anchored at time t.

Example 7. Figure 6 displays s and, for each time point t, the pattern dissimilarity of the pattern anchored at $r_1(t)$ to the query pattern $P(840)$, i.e., $\delta(P(t), P(840))$. Figure 6a does this for pattern length $l = 1$. The pattern dissimilarity is zero whenever the value of s is equal to $s(840)$. Figure 6b shows what happens if we increase the pattern length to $l = 60$. Also for this case whenever the pattern dissimilarity for the reference time series r_1 is zero, i.e., for 480 and 120, we have value 0.86 for time series s. Observe that for increasing values of l less patterns with distance zero exist (e.g., two in Fig. 6b instead of 5 in Fig. 6a). But the patterns with $l > 1$ at distance zero describe the situation better: $s(840)$ is located at a down-slope, and in Fig. 6b values of s where the pattern distance is zero only exist at down-slopes, while in Fig. 6a we have such values at both up- and down-slopes. □

Example 8. For shifted time series a pattern length $l > 1$ in addition captures the trend of time series and yields a more accurate imputation. First, Figure 7a illustrates s and the pattern dissimilarity to r_2 for $l = 1$. Figure 7b shows the same setting with $l = 60$. With $l > 1$ the pattern dissimilarity reaches zero only for time points 480 and 120, where time series s has value 0.86, which is the expected value for missing value $s(840)$. This illustrates that by increasing l, TKCM finds anchor points where the incomplete

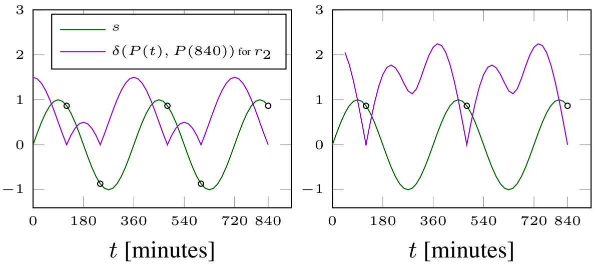


(a) Pattern length $l = 1$

(b) Pattern length $l = 60$

Figure 6: A longer pattern reduces the number of patterns that are identical to the query pattern.

time series s has similar values and trends. Consequently, TKCM uses pattern length l to effectively deal with shifted time series that are not linearly correlated. □



(a) Pattern length $l = 1$

(b) Pattern length $l = 60$

Figure 7: For shifted time series a longer pattern finds historical situations that are similar in value and trend.

LEMMA 5.1. *(*Monotonicity in Pattern Length*) The number of patterns that are within a distance τ to query pattern $P(t_n)$ decreases as the pattern length l increases:*

$$|\{t_m|\delta(P_{l+1}(t_m), P_{l+1}(t_n)) \le \tau\}| \le |\{t_m|\delta(P_l(t_m), P_l(t_n)) \le \tau\}|$$

PROOF. Let $\Delta = \delta(P_l(t_m), P_l(t_n))$. If pattern length l is increased we get $\delta(P_{l+1}(t_m), P_{l+1}(t_n)) = \sqrt{\Delta^2 + \sum_{r_i \in \mathbf{R}_s}(r_i(t_{m-l}) - r_i(t_{n-l}))^2}$. Observe that both terms under the square root are non-negative, hence δ is monotonically increasing as l increases. It follows that the number of patterns with distance $\le \tau$ decreases as l grows. □

5.3 Consistent Imputation

Definition 5. (Pattern-Determining at time t_n) At time t_n the reference time series $\mathbf{R}_s$ *pattern-determine* time series s, written $\mathbf{R}_s, t_n \xrightarrow{\text{pd}} s$, if for the k most similar anchor points $\mathbf{A}$ (cf. Def. 3) and patterns of length l the following holds for a small value ϵ:

$$\forall t_i, t_j \in \mathbf{A} : |s(t_i) - s(t_j)| \le \epsilon$$

Example 9. In our running example (cf. Fig. 3), the query pattern $P(t_n{=}14{:}20)$ is based on the two reference time series $\mathbf{R}_s = \{r_1, r_2\}$. The $k = 2$ most similar anchor points are $\mathbf{A} = \{14{:}00, 13{:}35\}$, and the values of s at these two anchors are 21.9°C and 21.8°C, respectively. The two reference time series $\mathbf{R}_s$ pattern-determine s at time t_n (written $\mathbf{R}_s, 14{:}20 \xrightarrow{\text{pd}} s$) with $\epsilon = |21.9°\text{C} - 21.8°\text{C}| = 0.1°\text{C}$. □

Pattern-determining time series guarantee that for a missing value $s(t_n)$ we find in the sliding window at least k similar patterns $P(t_{i_1}), \dots, P(t_{i_k})$ to $P(t_n)$ and the missing value $s(t_n)$ is similar to the observed values $s(t_i)$.

Definition 6. (*Consistent Time Series*) Let s be a time series with missing value $s(t_n) = \text{NIL}$. Let $\hat{s}$ be a time series where the missing value $s(t_n)$ has been imputed, that is $\hat{s}(t_n) \neq \text{NIL}$ and $\forall t \in W \setminus \{t_n\} : \hat{s}(t) = s(t)$. Time series $\hat{s}$ is *consistent* if $\forall t \in \mathbf{A} : |\hat{s}(t) - \hat{s}(t_n)| \le \epsilon$.

When TKCM imputes an incomplete time series s, we get an imputed time series $\hat{s}$. Intuitively, $\hat{s}$ is *consistent* if its value at the current time t_n is similar to past values when the reference time series observed a similar pattern.

LEMMA 5.2. *Let s be an incomplete time series with a missing value $s(t_n) = \text{NIL}$ and $\mathbf{R}_s$ its reference time series. Let $\hat{s}$ be the imputed time series produced by TKCM. If (a) $\mathbf{R}_s, t_n \xrightarrow{\text{pd}} s$ and (b) $s(t_n)$ is imputed as defined in Eq. 4, $\hat{s}$ is a consistent time series.*

PROOF. Let $P(t_n)$ be the query pattern that TKCM constructs for the reference time series in $\mathbf{R}_s$ with pattern length l. TKCM looks for the k most similar anchor points $\mathbf{A}$ with respect to $P(t_n)$. Since the reference time series $\mathbf{R}_s$ pattern-determine s at time t_n, we have that $\forall t, t' \in \mathbf{A} : |s(t) - s(t')| \le \epsilon$. The imputed value $\hat{s}(t_n)$ is the average of s at the anchor points (cf. Eq. 4). Since all values of s at $\mathbf{A}$ are similar among each other within a distance of ϵ, their mean $\hat{s}(t_n)$ is equally similar within an ϵ distance to all of them, i.e. $\forall t \in \mathbf{A} : |\hat{s}(t) - \hat{s}(t_n)| \le \epsilon$. Consequently the imputed time series $\hat{s}$ is consistent. □

Next, we give an example of pattern-determining time series to illustrate what kind of phenomena TKCM can handle. Specifically, we show that sine waves of the form $f(t) = A \times \text{sind}(t\frac{360}{P} + \phi) + o$ with amplitude A, period P, offset o, and phase shift ϕ are pattern-determining and that TKCM achieves consistent imputation on these series. The experiments in Section 7 confirm that this also holds for real world time series.

LEMMA 5.3. *Assume $s(t) = A_1 \times \text{sind}(t\frac{360}{P} + \phi_1) + o_1$ and $r(t) = A_2 \times \text{sind}(t\frac{360}{P} + \phi_2) + o_2$. Then $\mathbf{R}_s = \{r\}$ is pattern-determining s at time t_n for $l > 1$, $k \ge 1$ and $L \ge kP + l$.*

PROOF. Observe that for $l > 1$, pattern $P(t_n)$ occurs exactly once every full period, i.e., $P(t) = P(t_n)$ only if $t = t_{n-iP}$ for every $i \in \mathbb{N}$. Since $L \ge kP + l$ we know there are k patterns $P(t)$, such that $P(t) = P(t_n)$. Since s has the same periodicity as r, we know that $\forall t, t' \in \mathbf{A} : |s(t) - s(t')| \le 0 = \epsilon$. □

6. IMPLEMENTATION OF TKCM

6.1 Overview

To impute a missing value in a time series s at the current time t_n, TKCM performs three steps:

1. *Pattern Extraction:* Extract the anchor points of all candidate patterns from the streaming window W and compute the dissimilarity of these patterns to query pattern $P(t_n)$ (cf. Definition 2).

2. *Pattern Selection:* Select from the anchor points determined in step 1 the subset $\mathbf{A}$ of the time points that anchor the k most similar non-overlapping patterns (cf. Definition 3).

3. *Value Imputation:* Impute the missing value at t_n using anchor points $\mathbf{A}$ (cf. Definition 4).

In step 2, TKCM must find the k patterns that neither overlap each other nor $P(t_n)$, and that minimize the sum of dissimilarities with respect to $P(t_n)$. A simple greedy algorithm that sorts the anchor points according to dissimilarity and picks the first k ones that do not overlap fails to minimize the sum of dissimilarities. Therefore, we propose a dynamic programming scheme that exploits an optimal sub-structure of this problem. Let $D[j]$ denote the dissimilarity between the jth pattern in the window and the query pattern, and $M[i,j]$ denote the sum of dissimilarities of the i most similar non-overlapping patterns from among the first j patterns in W, where $i \leq j$. $M[i,j] = 0$ if $i = 0$, because no patterns have to be chosen. Similarly, $M[i,j] = \infty$ if $i > j$ because we cannot possibly find i non-overlapping patterns if we have only $j < i$ to choose from. Otherwise, we have two options: either (a) we omit the jth pattern and pick i patterns that possibly overlap it; or (b) we pick the jth pattern having dissimilarity $D[j]$ and have space left for $i{-}1$ patterns that do not overlap it. In the latter case, the first pattern to no longer overlap the jth pattern, if one exists, is the $(j-l)$th pattern.

This yields the following recurrence that minimizes the sum of dissimilarities:

$$M[i,j] = \begin{cases} 0 & \text{if } i = 0, \\ \infty & \text{if } i > j, \\ \min \begin{cases} M[i,j{-}1] \\ D[j] + M[i{-}1, j-l] \end{cases} & \text{otherwise} \end{cases} \tag{5}$$

The sum of dissimilarities of the k most similar anchor points is given by $M[k, L{-}2l{+}1]$, since $L{-}2l{+}1$ is the number of anchor points when we exclude the $l-1$ first and l last time points in W (cf. Def. 3).

6.2 Algorithm

The implementation uses one ring buffer of length L for each time series s and an offset O into the ring buffers to efficiently update the streaming window. The value at time t_n is located at $s[O]$ and the oldest value at $s[(O{+}1)\%L]$, where % is the modulo operator. TKCM's pseudo code is listed in Algorithm 1. The input parameters are the ring buffer for the incomplete time series s with a missing value at $s[O]$, d ring buffers R for the reference time series, the window size L, the pattern length l, and the number of anchor points k. The algorithm stores the imputed value in $s[O]$ and returns it.

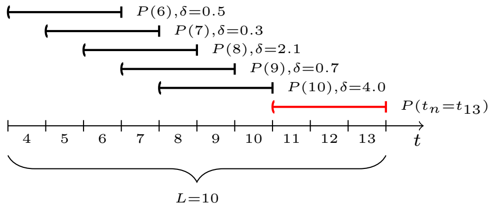
For processing, the algorithm uses an array D to store pattern dissimilarities, a $(k{+}1) \times (L{-}2l{+}2)$ matrix M to store the result for the dynamic programming algorithm, and an array A of size k to store the k most similar anchor points.

Lines 1–7 correspond to step 1, where the dissimilarities of all patterns in W are computed and stored in array D. The first $l{-}1$ and the last l time points are ignored as described above. In step 2, the algorithm finds the k most similar anchor points (Lines 8–23). Lines 8–14 implement the recurrence in Equation 5. $\max(j{-}l, 0)$ computes the predecessor of the jth pattern, yielding $j = 0$ if no such predecessor exists. Once matrix M is filled, $M[k, L{-}2l{+}1]$ contains the sum of dissimilarities of the k most similar anchor points $\mathbf{A}$. Finally, TKCM backtracks in lines 15–23 to find the anchor points $\mathbf{A}$. The algorithm starts in the lower-right most cell $M[k, L{-}2l{+}1]$ and applies the recurrence backwards. If for a cell $M[i,j]$ we have $M[i,j] = M[i,j{-}1]$ the jth anchor point is skipped as it is not part of the optimal solution, and the algorithm proceeds at cell $M[i,j{-}1]$. Otherwise, the jth anchor point is added to $\mathbf{A}$ and the algorithm proceeds with cell $M[i{-}1, \max(j{-}l, 0)]$ until i reaches 0, indicating that k anchor points have been chosen. Finally, in lines 24–27, which correspond to step 3, the algorithm imputes the missing value using the k most similar anchor points according to Definition 4.

Algorithm 1: TKCM

```
   Input: Ring buffer $s$, Array $R$ of $d$ ring buffers, window size $L$,
          pattern length $l$, and $k$.
   Output: Imputed value in $s[O]$.
1  for $j \leftarrow 1$ to $L{-}2{*}l{+}1$ do
2      $D[j] \leftarrow 0$;
3      for $i \leftarrow 0$ to $d{-}1$ do
4          for $x \leftarrow 0$ to $l{-}1$ do
5              $y \leftarrow l{+}j{-}x{-}1$;
6              $D[j] \leftarrow D[j]{+}(R[i][(O{+}y)\%L]{-}R[i][(O{-}x)\%L])^2$;
7      $D[j] \leftarrow$ sqrt($D[j]$);
8  for $j \leftarrow 0$ to $L{-}2{*}l{+}1$ do
9      $M[0][j] \leftarrow 0$;
10     for $i \leftarrow 1$ to $k$ do
11         if $i > j$ then
12             $M[i][j] \leftarrow \infty$;
13         else
14             $M[i][j] \leftarrow \min(M[i][j{-}1], D[j]{+}M[i{-}1][\max(j{-}l, 0)])$;
15 $i \leftarrow k$;
16 $j \leftarrow L{-}2{*}l{+}1$;
17 while $i > 0$ do
18     if $M[i][j] = M[i][j{-}1]$ then
19         $j{-}{-}$;
20     else
21         $A[i{-}1] \leftarrow j$;
22         $i{-}{-}$;
23         $j \leftarrow \max(j{-}l, 0)$;
24 $s[O] \leftarrow 0$;
25 for $i \leftarrow 0$ to $k{-}1$ do
26     $s[O] \leftarrow s[O] + (s[(O{+}l{+}A[i]{-}1)\%L]/k)$;
27 return $s[O]$;
```

Example 10. The top of Fig. 8 shows a streaming window of length $L = 10$ with current time $t_n = t_{13}$. The query pattern of length $l = 3$ and all extracted patterns are shown as red and black intervals, respectively. The first (i.e., $j = 1$) pattern is $P(t_6)$ and the last pattern is $P(t_{10})$. The lower left table lists the patterns in the streaming window, their index j, predecessor, and dissimilarity with respect to query pattern $P(t_{13})$. For instance, $P(t_8)$ with index $j = 3$ has no non-overlapping predecessor, hence $\max(j-l, 0) = 0$. The right table shows the matrix M computed by Algorithm 1. For instance, $M[1,1]$ is the result of computing $\min(D[1] + M[1{-}1, \max(1{-}3, 0)], M[1, 1{-}1]) = \min(0.5{+}0, \infty) = 0.5$. $M[2,5] = 1.2$ in the lower-right corner contains the minimum sum of dissimilarity. To retrieve the k most similar non-overlapping patterns, the algorithm starts at $M[2,5]$ and follows the highlighted path through the matrix: gray means that a pattern was omitted and green means that a pattern is part of the final result. For instance, $M[2,5]$ is equal to $M[2,4]$, hence the $j = 5$th pattern $P(t_{10})$ is not part of the solution. $M[2,4]$, in turn, is the result of $D[4]{+}M[1,1] = 0.7{+}0.5 = 1.2$, hence $j = 4$th pattern $P(t_9)$ is part of the solution. Continuing with $M[1,1]$ we find another match before reaching $M[0,0]$. The algorithm finds the $k = 2$ anchor points $\mathbf{A} = \{t_6, t_9\}$ and computes the imputed value $\hat{s}(t_{13}) = {}^1\!/_2(s(t_6) + s(t_9))$.



$P(t)$	j	$\max(j-l, 0)$	$D[j]$
$P(t_6)$	1	0	0.5
$P(t_7)$	2	0	0.3
$P(t_8)$	3	0	2.1
$P(t_9)$	4	1	0.7
$P(t_{10})$	5	2	4.0

	$j=0$	1	2	3	4	5
$i=0$	0	0	0	0	0	0
1	∞	0.5	0.3	0.3	0.3	0.3
2	∞	∞	∞	∞	1.2	1.2

Figure 8: Dynamic programming algorithm to compute the time points of the top $k = 2$ non-overlapping patterns of length $l = 3$ that minimize the sum of dissimilarities.

6.3 Complexity Analysis

LEMMA 6.1. *When the current time t_n advances, TKCM needs $O(1)$ time per stream s to update the corresponding ring buffer of size $O(L)$.*

PROOF. When t_n advances, a new value replaces an old value in the time series, requiring $O(1)$ time in a ring buffer of size L. □

LEMMA 6.2. *The time complexity of TKCM to impute a missing value is $O((l \times d + k) \times L)$.*

PROOF. Initially TKCM computes the dissimilarity of $O(L)$ patterns, each of size $l \times d$, having an overall time complexity of $O(l \times d \times L)$. Next the algorithm iterates over the $O(k \times L)$ sized dynamic programming matrix M. Hence the overall time complexity is $O(l \times d \times L + k \times L)$. □

LEMMA 6.3. *The space complexity of TKCM to impute a missing value is $O(k \times L)$.*

PROOF. The pattern extraction phase requires $O(L)$ space to store the dissimilarities of all patterns. TKCM needs $O(k \times L)$ space for matrix M. □

7. EXPERIMENTAL EVALUATION

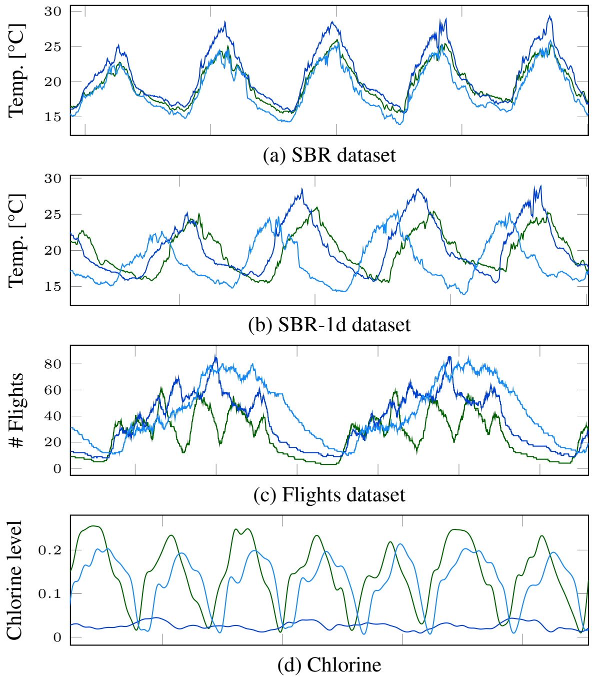
In the experiments we simulate large blocks of consecutively missing values (e.g. one week). We repeatedly call TKCM to impute each missing value. This simulates a common sensor failure that requires a technician to reach a faulty weather station and replace the broken sensor. As accuracy measure we use the root mean square error (RMSE), defined as

$$\text{RMSE} = \sqrt{\frac{1}{|T|} \sum_{t_n \in T} (s(t_n) - \hat{s}(t_n))^2},$$

where T is the set of missing time points. The experiments are conducted on a Linux server, running Ubuntu 14.04 server edition. It is powered by an Intel Xeon X5650 CPU with a frequency of 2.67GHz and 24GB of main memory. TKCM is implemented in C and compiled with Clang 3.4-1, based on LLVM 3.4.

7.1 Datasets and Setup

We use both real-world and synthetic datasets in our experimental evaluation. First, we use the SBR dataset of meteorological time series in South Tyrol (cf. Sec. 1). Second, we shift the time series of the SBR data set by a (different) random amount up to one day and call this dataset SBR-1d. Third, we use the Flights dataset [3] that consists of eight time series, each of length 8801 (6 days). A time series describes at time t the number of airplanes that departed from a given airport and are in the air at time t. Fourth, we use the publicly available Chlorine dataset [1] used by SPIRIT [16]. This synthetic dataset was generated by a simulation of a drinking water distribution system; it describes the chlorine concentration at 166 junctions over a time frame of 4310 time points (15 days) with a sample rate of 5 minutes. The propagation of the chlorine level in the system causes phase shifts in the dataset. Fig. 9 shows an excerpt of three sample time series from each dataset. Each time series has different amplitudes, phase shifts, and trends.



(a) SBR dataset

(b) SBR-1d dataset

(c) Flights dataset

(d) Chlorine

Figure 9: Three sample time series from each dataset.

We compare TKCM to three competitors: CD [13] (provided by the author), MUSCLES [25] (implemented in Matlab), and SPIRIT [23] (obtained from [1], Matlab code). SPIRIT's Matlab code does not impute missing values, hence we extended the code to use one autoregressive model per hidden variable as described in [23]. SPIRIT automatically adds or removes hidden variables as the streams evolve. When a hidden variable appears, a new autoregressive model of order $p = 6$ needs to be fitted, which requires at least p values of the new hidden variable before it can be used. If in that time a value needs to be imputed, the model is not yet ready. Consequently we fixed the number of hidden variables at two, which gave generally the best results in our experiments. For MUSCLES and SPIRIT we use a tracking window size of $p = 6$ as recommended by the authors [23, 25]. Contrary to the author's recommendation we set the exponential forgetting factor λ to 1 rather than to $0.96 \leq \lambda \leq 0.98$. We found that for $\lambda < 1$ the accuracy decreases, because the algorithms "forget" the old non-imputed (and accurate) values and adapt more to the new imputed (and inaccurate) values. CD has no parameters to tune. The code for our MUSCLES, SPIRIT, and TKCM implementations is available online[1].

[1] http://www.ifi.uzh.ch/en/dbtg/Staff/wellenzohn/dasa.html

7.2 Calibration

We begin with an initial calibration of TKCM's parameters d, k, and L. Unless otherwise noted we set the parameters to the following default values; $d = 3$ reference time series, $k = 5$ most similar anchor points, a streaming window of $L = 1$ year, and pattern length $l = 72$.

In the left column of Fig. 10 we show TKCM's accuracy for increasing values of d on three datasets; for brevity we omit the SBR dataset as it shows identical behavior to the SBR-1d dataset. TKCM's accuracy significantly increases (that is the RMSE decreases) as d increases up to $d = 3$ reference time series, while $d > 3$ does not provide significantly better accuracy. Since the Flights dataset has only 8 time series, we can set d at most to 7. The right column of Fig. 10 shows the impact of parameter k on TKCM's accuracy. In general we tend to pick small values of k, e.g., $k \in [3, 10]$ to get the best possible (most similar) patterns from the window. Larger values of k may add less similar patterns, in particular for short streaming windows. We observed that for the two small datasets Flights and Chlorine $k = 5$ is sufficient. TKCM's accuracy noticeably decreases on the Flights dataset for $k > 5$, because the dataset contains only measurements for 6 days and if one day is missing we try to find more than 5 similar situations within 5 days, which makes no sense. For the larger (1 year) SBR and SBR-1d datasets we found that there is a marginal accuracy increase even from $k = 5$ to $k = 10$, after which the accuracy remains stable. Therefore, our recommendation is to use a small value of k, e.g. $k = 5$, and if the dataset is large, one can safely double k.

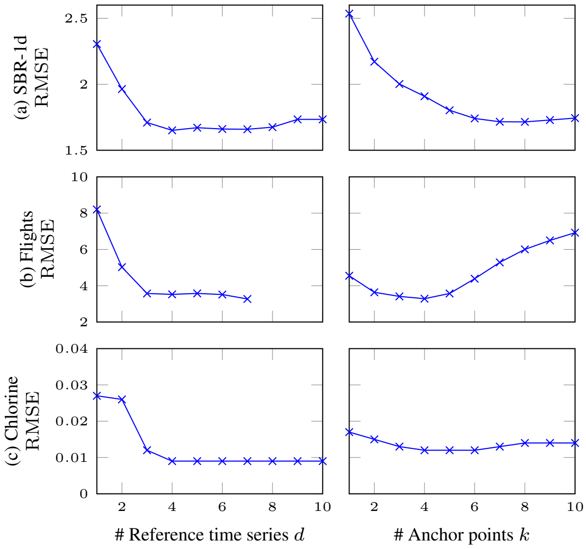


Figure 10: Calibration shows that $d = 3$ and $k = 5$ are good default values.

For the small Flights and Chlorine datasets (6 and 15 days, respectively) we use in our experiments the entire time range as streaming window length L. For the SBR and SBR-1d datasets we use a streaming window length $L = 105120$ (1 year), because one year covers the whole temperature range and contains each pattern several times. This choice is conservative; we found that already a window size of 6 months gives a good accuracy of RMSE = 1.8°C, which only dropped to 1.7°C for a 5 year window. In general, larger window sizes L provide only a slightly superior accuracy.

7.3 Accuracy

7.3.1 Pattern Length l

For shifted time series, the pattern length l is the key to an accurate and robust imputation. In Fig. 11, we evaluate l by varying the pattern length l from 1 to 144 (i.e., a pattern that spans 12 hours). As expected, for the (non-shifted) SBR dataset l has close to no impact on TKCM's accuracy, because there is a high linear correlation between the incomplete time series and its $d = 3$ reference time series. For the SBR-1d dataset the RMSE drops by about 0.5°C (25%) by increasing l to 72. On the flight dataset we observe an improvement of 50% for $l = 72$ and 60% for $l = 144$. The reason why we see an improvement beyond $l = 72$ is the different sample rate of 1 minute of the Flights dataset as opposed to the 5 minutes in the SBR dataset. While $l = 72$ yields a pattern that spans 6 hours in the SBR dataset, it only spans 1 hour in the Flights dataset. On the Chlorine dataset we observe an accuracy increase of 60% with pattern length $l = 72$, after which the accuracy slightly decreases.

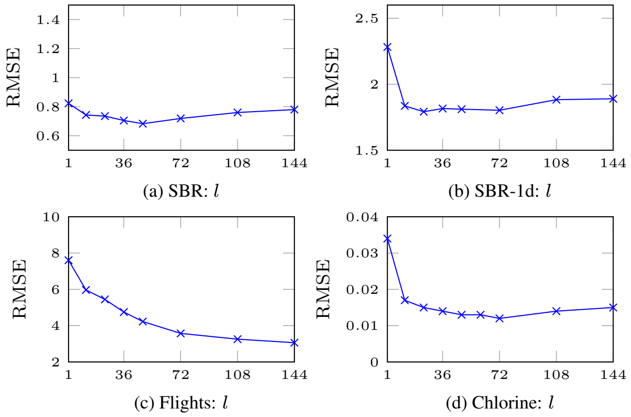


Figure 11: Pattern length l evaluated on each dataset.

To put these raw numbers into perspective and see the real impact of l, we compare TKCM's recovery of an incomplete time series s in Fig. 12 with pattern length $l = 1$ (left column) and $l = 72$ (right column). Observe how much TKCM's recovery oscillates with $l = 1$ and how well TKCM adapts to the assumed missing time series s with $l = 72$. Even for the SBR dataset there is a slight oscillation, albeit minimal compared to the three shifted datasets. The reason for this strong oscillation when $l = 1$ is that with a short pattern, the reference time series do not pattern-determine the incomplete time series. The difference in dissimilarity between "good" patterns and "bad" patterns is too small for TKCM to detect, as explained in Sec. 5.1. Put differently, in the presence of shifts, time series are no longer linearly correlated; whenever a reference time series observes a very similar value, the incomplete time series has very different values.

Fig. 13a shows the scatterplot of an incomplete time series s in the Chlorine dataset against one of its reference time series r_1. There is clearly no strong linear correlation ($\rho(s, r_1) = 0.5$): e.g. for $r_1(t) = 0.1$, $s(t)$ has two different values (0 and 0.15). Fig. 13b shows that the average ϵ (cf. Def. 5) decreases as l increases on the Chlorine dataset with $k = 5$. Value $\epsilon = \max_{t,t' \in \mathbf{A}} |s(t) - s(t')|$ essentially describes the range of the values of s at the k most similar anchor points $\mathbf{A}$. The lower ϵ gets, the less the values of s differ at the most similar anchor points, which indicates that the reference time series strongly pattern determine s for k and l. Notice that the average ϵ increases after $l = 72$, which

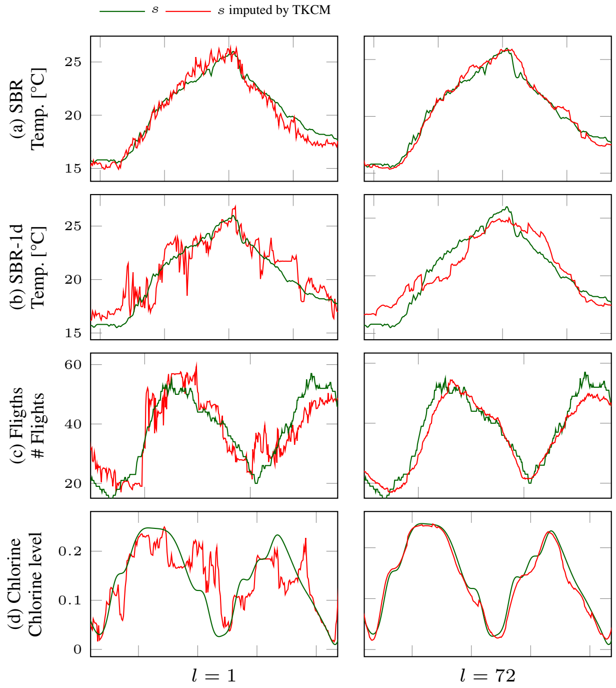
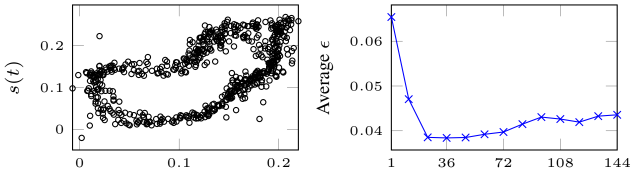


Figure 12: The reason for the strong oscillation in TKCM's recovery with $l = 1$ are shifts in the reference time series. Increasing the pattern length l helps TKCM to detect shifts.

coincides with our observations in Fig. 11d.



(a) $r_1(t)$ (b) Pattern length l

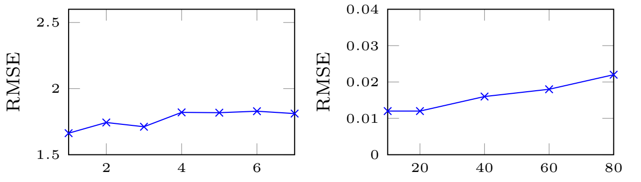
Figure 13: *Left:* Scatterplot of s against the reference time series r_1. *Right:* Range of s at the k anchor points (Chlorine dataset).

7.3.2 *Missing Block Length*

In Fig. 14 we study TKCM's accuracy in terms of the length of the missing block, i.e., the number of consecutively missing values that TKCM needs to impute. First, we use our large dataset SBR-1d and simulate sensor failures of up to several weeks. Fig. 14a shows that the accuracy of TKCM only slightly decreases by 0.2°C as the block length grows from 1 to 4 weeks, after which the accuracy plateaus. Next, we increase the missing block length for the small dataset Chlorine from 10% to 80% of the dataset size. We start with the remaining 90% to 20% of the dataset in the streaming window of length $L = 4310$ and impute the rest of the dataset as missing values. Fig. 14b shows that also for this case the accuracy decreases only slowly.

7.3.3 *Comparison with Competitors*

SBR. We first perform a baseline comparison of all algorithms on the SBR dataset that has no phase shifts. Fig. 15a shows an excerpt of the experiment where we assume a block of values is missing in time series s, and impute the missing values with each approach. TKCM, SPIRIT, MUSCLES, and CD perform virtually equally well. Observe that the last valley in the temperature curve shows a higher temperature than the previous valleys. While TKCM and CD are able to capture this trend, MUSCLES and SPIRIT impute a too low value. The most likely reason is that the models that MUSCLES and SPIRIT build are not able to adapt quickly enough to the new behavior of the time series, while TKCM adapts instantaneously to the changing behavior.



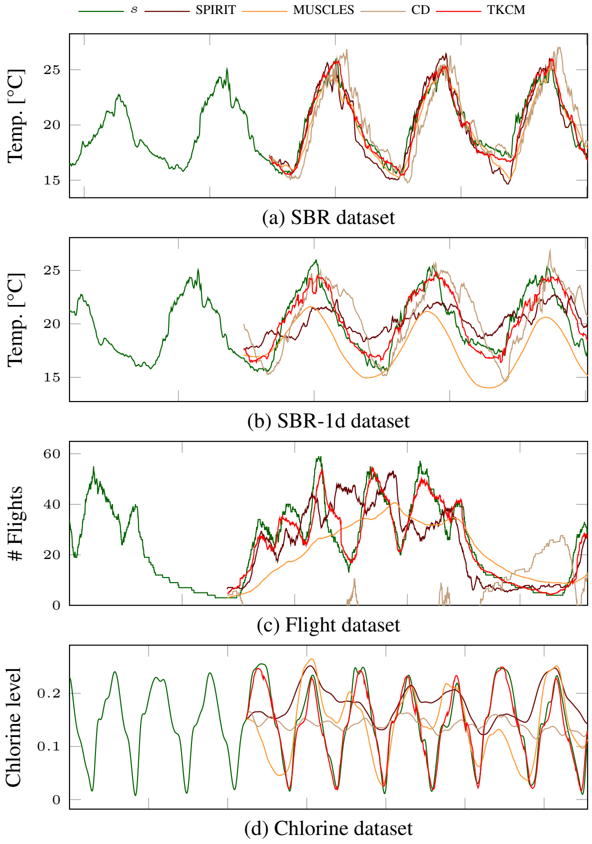
(a) Block length [weeks] (b) Block length [% of L]

Figure 14: Impact of the missing block length on the accuracy (Chlorine dataset).

SBR-1d. Next we impute the same block of missing values in the SBR-1d dataset that has shifted time series. Observe how TKCM's accuracy only slightly decreases in Fig. 15b; TKCM slightly misses the second last downwards slope, but the last valley is again accurately imputed. SPIRIT completely misses the amplitude; its recovered peaks are too low and its valleys are too high in temperature. Moreover, the overall trend of the missing block is not well recovered. MUSCLES recovery is borderline at best; the overall periodicity of the signal is recovered, but MUSCLES was not able to recover any feature of s. Moreover, s has a slightly increasing temperature trend, but MUSCLES' recovery has a decreasing trend. CD's recovery is shifted with respect to s, as also indicated by the discontinuous imputation at the very beginning of the missing block.

Flights. On the Flights dataset we observe a similar behavior as in the previous experiment. Fig. 15c shows that TKCM captures each peak and valley accurately, while SPIRIT's accuracy decreases over time. Initially, the trend of s is vaguely captured, but after the highest peak, the trend of SPIRIT's imputation is *inverse* (i.e., peaks and valleys are swapped) with respect to the true signal. Again, MUSCLES produces an extremely smoothed signal, that does not resemble the true time series; peaks and valleys are not recovered. CD's recovery is only partially shown as many recovered values are *negative*. Most likely, the large block of missing values (ca. 20% of the dataset) is the reason.

Chlorine. In the Chlorine dataset TKCM captures the trend of s generally well, the valleys are almost perfectly recovered, while the peaks are slightly less accurate. SPIRIT's recovery does not capture the amplitude of s, and also the trend of the recovery does not match that of s. MUSCLES completely misses the first peak, imputing it with a valley instead; also the general trend of MUSCLES' recovery does not resemble s. In this dataset we found MUSCLES and SPIRIT to perform with widely differing accuracies, sometimes their imputations is good, sometimes worse than in this example. There is no clear pattern when either approach works or fails. CD's recovered signal has a very small amplitude and also the trend does not capture that of s.



(a) SBR dataset

(b) SBR-1d dataset

(c) Flight dataset

(d) Chlorine dataset

Figure 15: Imputation of incomplete time series s with different imputation techniques on four different datasets.

Summary. Fig. 16 shows the RMSE for all compared algorithms on each dataset. In this comparison we impute 4 time series per data set; with missing block lengths per time series of 1 week in the SBR and SBR-1d datasets, and 20% of the dataset size for Flights and Chlorine. All algorithms are given the same amount of data (L measurements per time series). We use $L = 6$ months for the SBR and SBR-1d datasets, because of CD's prohibitively large runtime for $L = 1$ year (our default) and the default L for the remaining datasets. The experiments show that only for the non-shifted SBR dataset all algorithms provide a comparable accuracy. For the remaining three shifted datasets, TKCM clearly outperforms its competitors both in terms of perceived accuracy (Fig. 15) and raw RMSE (Fig. 16). Our general observation is that non-shifted linearly-correlated data poses no significant challenge to any algorithm. As soon as shifts are present in the data, the accuracy of state-of-the-art solutions is largely unpredictable, ranging from good to unusable.

7.4 Runtime

As discussed in Sec. 6.3, TKCM's time complexity is linear with respect to all parameters (l, d, k, and L) as confirmed by Fig. 17. In this experiments on the SBR-1d dataset we vary each parameter, leaving the other three parameters at their defaults ($l = 72$, $d = 3$, $k = 5$ and $L = 1$ year). Fig. 17a and Fig. 17b show TKCM's runtime with respect to the size of the query pattern $P(t_n)$, Fig. 17c shows the impact of the number of anchor points k, and Fig. 17d shows the impact of the streaming window size L on the runtime. Parameter L has the largest impact on TKCM's runtime, followed by l and d with similar impact. Parameter k is relatively cheap – even if set to very large values, e.g., $k > 50$. For our default parameter settings we observe a runtime of approximately 2 seconds to impute a single missing value.

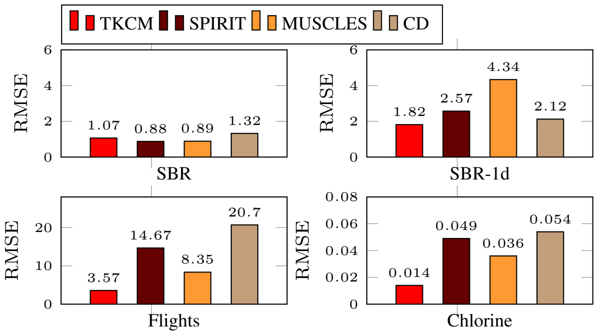
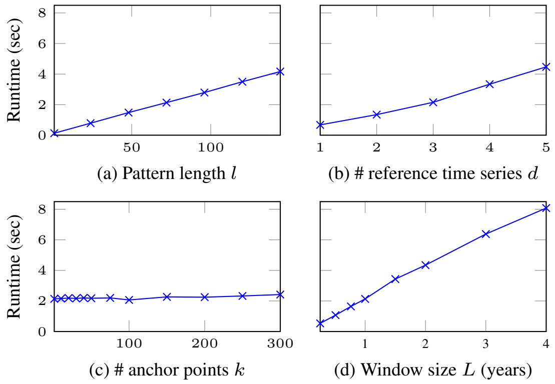


Figure 16: Comparison of TKCM, SPIRIT, MUSCLES, and CD for each dataset.



(a) Pattern length l

(b) # reference time series d

(c) # anchor points k

(d) Window size L (years)

Figure 17: Runtime experiments. TKCM's time complexity is linear with respect to all its parameters l, d, k, and L (SBR-1d dataset).

Performance breakdown. As described in Sec. 6 the two main phases of TKCM are *pattern extraction* (PE) and *pattern selection* (PS). In our default setup, the PE-phase accounts for 92% of TKCM's overall runtime. If we further subdivide the PE-phase, we see that 82% of the overall runtime are required to fetch data from main memory and 10% are used to compute the pattern dissimilarity δ. If we increase k to 300 we see the runtime of the PS-phase climbing from 8% up to 25%. Thus, for the default value of k, the runtime incurred by the PS-phase is outweighed by the PE-phase. Hence, to improve TKCM's performance, future research must focus on speeding up the pattern extraction phase.

Comparison. A direct comparison of the runtimes of the considered approaches is not meaningful, because the systems are implemented in different programming languages (TKCM in C, CD in Java, MUSCLES and SPIRIT in Matlab). To give a rough feeling for the overall performance we consider each approach in turn. CD is an offline algorithm and not applicable to streams. CD's matrix decomposition lasted in our experiments roughly 20 minutes per execution and is hence not applicable to streaming environments. Both SPIRIT and MUSCLES required one millisecond to impute one missing value, TKCM requires roughly 2 seconds.

8. CONCLUSION AND FUTURE WORK

We studied the problem of missing values in meteorological streams of time series data and presented an algorithm, termed TKCM, to accurately impute missing values in a streaming environment. If the current value in a time series s is missing, TKCM determines a two-dimensional query pattern over the last l measurements of d reference time series. It then retrieves the anchor points of the k most similar non-overlapping patterns to the query pattern. The missing value is computed from the values of s at these k anchor points. We show that TKCM achieves consistent imputation if the reference time series pattern-determine s, which covers non-linear relationships between time series such as phase shifts. An extensive experimental evaluation using four real-world and synthetic datasets confirms that TKCM is accurate and outperforms state-of-the-art competitors.

Future work points in several directions. First, we will work on the efficiency of TKCM, in particular the pattern extraction phase, which proved to be the most time-consuming component. In particular, we plan to reduce the number of extracted patterns by pruning patterns that cannot possibly belong to an optimal solution. Second, we plan to investigate how to automatically determine the best candidate reference time series, although in many application domains (and especially in meteorology) we can rely on human experts. Third, we plan to compare different dissimilarity functions δ (e.g. L_1-norm, DTW [9], etc.). Moreover, it would be interesting to compute an alignment between shifted time series (e.g., using DTW [9]) and to compare TKCM's accuracy on the aligned time series using a pattern length $l = 1$ to the accuracy on the shifted time series using $l > 1$.

9. ACKNOWLEDGMENTS

The work has been done as part of the DASA project, which is funded by the Foundation of the Free University of Bozen-Bolzano. We wish to thank our partners at the Südtiroler Beratungsring and the Research Centre for Agriculture and Forestry Laimburg for the good collaboration and helpful domain insights they provided, in particular Armin Hofer, Martin Thalheimer, and Robert Wiedmer. We also want to thank Mourad Khayati for his input and for sharing his CD implementation used in the experimental evaluation. We thank the anonymous reviewers for their valuable comments and suggestions.

10. REFERENCES

[1] SPIRIT project. https://www.cs.cmu.edu/afs/cs/project/spirit-1/www/. Accessed: 2016-05-29.

[2] G. E. A. P. A. Batista and M. C. Monard. An analysis of four missing data treatment methods for supervised learning. *Applied Artificial Intelligence*, 17(5-6), 2003.

[3] A. Behrend and G. Schüller. A case study in optimizing continuous queries using the magic update technique. In *SSDBM*, pages 31:1–31:4, 2014.

[4] G. E. P. Box and G. Jenkins. *Time Series Analysis, Forecasting and Control*. Holden-Day, Incorporated, 1990.

[5] B. Chiu, E. Keogh, and S. Lonardi. Probabilistic discovery of time series motifs. In *KDD*, pages 493–498, 2003.

[6] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. *SIGMOD Rec.*, 23(2):419–429, May 1994.

[7] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and information Systems*, 3(3):263–286, 2001.

[8] E. Keogh, J. Lin, and A. Fu. HOT SAX: Efficiently finding the most unusual time series subsequence. In *ICDM*, pages 226–233, 2005.

[9] E. J. Keogh. Exact indexing of dynamic time warping. In *VLDB*, 2002.

[10] E. J. Keogh and T. Rakthanmanon. Fast shapelets: A scalable algorithm for discovering time series shapelets. In *ICDM*, pages 668–676, 2013.

[11] M. Khayati and M. H. Böhlen. REBOM: recovery of blocks of missing values in time series. In *COMAD*, pages 44–55, 2012.

[12] M. Khayati, M. H. Böhlen, and J. Gamper. Memory-efficient centroid decomposition for long time series. In *ICDE*, pages 100–111, 2014.

[13] M. Khayati, P. Cudré-Mauroux, and M. H. Böhlen. Using lowly correlated time series to recover missing values in time series: a comparison between SVD and CD. In *SSTD*, pages 237–254, 2015.

[14] L. Li, J. McCann, N. S. Pollard, and C. Faloutsos. DynaMMo: Mining and summarization of coevolving sequences with missing values. In *KDD*, pages 507–516, 2009.

[15] P. Merlin, A. Sorjamaa, B. Maillet, and A. Lendasse. X-SOM and L-SOM: A double classification approach for missing value imputation. *Neurocomputing*, 73(7-9):1103–1108, 2010.

[16] S. Papadimitriou, J. Sun, and C. Faloutsos. Streaming pattern discovery in multiple time-series. In *VLDB*, pages 697–708, 2005.

[17] S. Papadimitriou, J. Sun, C. Faloutsos, and P. S. Yu. Dimensionality reduction and filtering on time series sensor streams. In *Managing and Mining Sensor Data*, pages 103–141. 2013.

[18] J. L. H. Paulhus and M. A. Kohler. Interpolation of Missing Precipitation Records. *Monthly Weather Review*, 80(8), Aug. 1952.

[19] D. Rubin. Multiple Imputation after 18+ Years. *Journal of the American Statistical Association*, 91(434), 1996.

[20] J. L. Schafer and J. W. Graham. Missing data: our view of the state of the art. *Psychological Methods*, 7, 2002.

[21] J. Shieh and E. Keogh. iSAX: Indexing and mining terabyte sized time series. In *KDD*, pages 623–631, 2008.

[22] A. Sorjamaa, P. Merlin, B. Maillet, and A. Lendasse. SOM+EOF for finding missing values. In *ESANN*, pages 115–120, 2007.

[23] J. Sun, S. Papadimitriou, and C. Faloutsos. Online latent variable detection in sensor networks. In *ICDE*, pages 1126–1127, 2005.

[24] O. G. Troyanskaya, M. N. Cantor, G. Sherlock, P. O. Brown, T. Hastie, R. Tibshirani, D. Botstein, and R. B. Altman. Missing value estimation methods for DNA microarrays. *Bioinformatics*, 17(6), 2001.

[25] B. Yi, N. Sidiropoulos, T. Johnson, H. V. Jagadish, C. Faloutsos, and A. Biliris. Online data mining for co-evolving time sequences. In *ICDE*, pages 13–22, 2000.

[26] C. Yozgatligil, S. Aslan, C. Iyigun, and I. Batmaz. Comparison of missing value imputation methods in time series: the case of turkish meteorological data. *Theoretical and Applied Climatology*, 112(1-2), 2013.

Data-driven Schema Normalization

Thorsten Papenbrock
Hasso Plattner Institute (HPI)
14482 Potsdam, Germany
thorsten.papenbrock@hpi.de

Felix Naumann
Hasso Plattner Institute (HPI)
14482 Potsdam, Germany
felix.naumann@hpi.de

ABSTRACT

Ensuring Boyce-Codd Normal Form (BCNF) is the most popular way to remove redundancy and anomalies from datasets. Normalization to BCNF forces functional dependencies (FDs) into keys and foreign keys, which eliminates duplicate values and makes data constraints explicit. Despite being well researched in theory, converting the schema of an existing dataset into BCNF is still a complex, manual task, especially because the number of functional dependencies is huge and deriving keys and foreign keys is NP-hard.

In this paper, we present a novel normalization algorithm called Normalize, which uses discovered functional dependencies to normalize relational datasets into BCNF. Normalize runs entirely data-driven, which means that redundancy is removed only where it can be observed, and it is (semi-)automatic, which means that a user may or may not interfere with the normalization process. The algorithm introduces an efficient method for calculating the closure over sets of functional dependencies and novel features for choosing appropriate constraints. Our evaluation shows that Normalize can process millions of FDs within a few minutes and that the constraint selection techniques support the construction of meaningful relations during normalization.

1. FUNCTIONAL DEPENDENCIES

A functional dependency (FD) is a statement of the form $X \rightarrow A$ with X being a set of attributes and A being a single attribute from the same relation R. We say that the left-hand-side (Lhs) X functionally determines the right-hand-side (Rhs) A. This means that whenever two records in an instance r of R agree on all their X values, they must also agree on their A value [7]. More formally, an FD $X \rightarrow A$ holds in r, iff $\forall t_1, t_2 \in r : t_1[X] = t_2[X] \Rightarrow t_1[A] = t_2[A]$. In the following, we consider only *non-trivial* FDs, which are FDs with $A \notin X$.

Table 1 depicts an example address dataset for which the two functional dependencies Postcode→City and Postcode→Mayor hold. Because both FDs have the same Lhs, we can aggregate them to the notation Postcode→City,Mayor. The presence of this FD introduces anomalies in the dataset, because the values `Potsdam`, `Frankfurt`, `Jakobs`, and `Feldmann` are stored redundantly and updating these values might cause inconsistencies. So if, for instance, some Mr. Schmidt was elected as the new mayor of Potsdam, we must correctly change all three occurrences of `Jakobs` to `Schmidt`.

Table 1: Example address dataset

First	Last	Postcode	City	Mayor
Thomas	Miller	14482	Potsdam	Jakobs
Sarah	Miller	14482	Potsdam	Jakobs
Peter	Smith	60329	Frankfurt	Feldmann
Jasmine	Cone	01069	Dresden	Orosz
Mike	Cone	14482	Potsdam	Jakobs
Thomas	Moore	60329	Frankfurt	Feldmann

Such anomalies can be avoided by normalizing relations into the Boyce-Codd Normal Form (BCNF). A relational schema R is in BCNF, iff for all FDs $X \rightarrow A$ in R the Lhs X is either a key or superkey [7]. Because Postcode is neither a key nor a superkey in the example dataset, this relation does not meet the BCNF condition. To bring all relations of a schema into BCNF, one has to perform six steps, which are explained in more detail later: (1) discover all FDs, (2) extend the FDs, (3) derive all necessary keys from the extended FDs, (4) identify the BCNF-violating FDs, (5) select a violating FD for decomposition (6) split the relation according to the chosen violating FD. The steps (3) to (5) repeat until step (4) finds no more violating FDs and the resulting schema is BCNF-conform. We find several FD discovery algorithms, such as Tane [14] and HyFD [19], that serve step (1), but there are, thus far, no algorithms available to efficiently and automatically solve the steps (2) to (6).

For the example dataset, an FD discovery algorithm would find twelve valid FDs in step (1). These FDs must be aggregated and transitively extended in step (2) so that we find, inter alia, First,Last→Postcode,City,Mayor and Postcode→City,Mayor. In step (3), the former FD lets us derive the key {First, Last}, because these two attributes functionally determine all other attributes of the relation. Step (4), then, determines that the second FD violates the BCNF condition, because its Lhs Postcode is neither a key nor superkey. If we assume that step (5) is able to automatically select the second FD for decomposition, step (6) decomposes the example relation into R_1(First, Last, Postcode) and R_2(Postcode, City, Mayor) with {First, Last} and {Postcode} being primary keys and R_1.Postcode→R_2.Postcode a foreign key constraint. Table 2 shows this result. When again checking for violating FDs, we do not find any and stop the nor-

©2017, Copyright is with the authors. Published in Proc. 20th International Conference on Extending Database Technology (EDBT), March 21-24, 2017 - Venice, Italy: ISBN 978-3-89318-073-8, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

Table 2: Normalized example address dataset

First	Last	Postcode
Thomas	Miller	14482
Sarah	Miller	14482
Peter	Smith	60329
Jasmine	Cone	01069
Mike	Cone	14482
Thomas	Moore	60329

Postcode	City	Mayor
14482	Potsdam	Jakobs
60329	Frankfurt	Feldmann
01069	Dresden	Orosz

malization process with a BCNF-conform result. Note that the redundancy in City and Mayor has been removed and the total size of the dataset was reduced from 36 to 27 values.

Because memory became a lot cheeper in the last years, there is a trend of not normalizing datasets for performance reasons. Normalization is, hence, today often claimed to be obsolete. This claim is false and ignoring normalization is dangerous for the following reasons [8]:

1. Normalization removes redundancy and, in this way, decreases error susceptibility and memory consumption. While memory might be relatively cheep, data errors can have serious and expensive consequences and should be avoided at all costs.

2. Normalization does not necessarily decrease query performance; in fact, it can even increase the performance. Some queries might need some additional joins after normalization, but others can read the smaller relations much faster. Also, more focused locks can be set, increasing parallel access to the data, if the data has to be changed. So the performance impact of normalization is not determined by the normalized dataset but by the application that uses it.

3. Normalization increases the understanding of the schema and of queries against this schema: Relations become smaller and closer to the entities they describe; their complexity decreases making them easier to maintain and extend. Furthermore, queries against the relations become easier to formulate and many mistakes are easier to avoid. For instance, aggregations over columns with redundant values are hard to formulate correctly.

In summary, normalization should be the default and denormalization a conscious decision, i.e., ”we should denormalize only at a last resort [and] back off from a fully normalized design only if all other strategies for improving performance have failed, somehow, to meet requiremnts“, C. J. Date, p. 88 [8].

The objective of this work is to normalize a given relational instance into Boyce-Codd Normal Form. Note that we do not aim to recover a certain schema nor do we aim to design a new schema using business logic. To solve the normalization task, we propose a data-driven, (semi-)automatic normalization algorithm that removes all FD-related redundancy while still providing full information recoverability. Being data-driven means that all FDs used in the normalization process are extracted directly from the data and that all decomposition proposals are based solely on data-characteristics. In other words, we consider only redundancy that can actually be observed in a given relational instance.

The advantage of a data-driven normalization approach over state-of-the-art schema-driven approaches is that it can use the data to expose all syntactically valid normalization options, i.e., functional dependencies with evidence in the data, so that the algorithm (or the user) must only decide for a normalization path and not find one. The number of FDs can, indeed, become large, but we show that an algorithm can effectively propose the semantically most appropriate options. Furthermore, knowing all FDs allows for a more efficient normalization algorithm as opposed to having only a subset of FDs.

Research challenges. In contrast to the vast amount of research on normalization in the past decades, we do not assume that the FDs are given, because this is almost never the case in practice. We also do not assume that a human data expert is able to manually identify them, because the search is difficult by nature and the actual FDs are often not obvious. The FD Postcode→City from our example, for instance, is commonly believed to be true although it is usually violated by exceptions where two cities share the same postcode; the FD Atmosphere→Rings, on the other hand, is difficult to discover for a human but in fact holds on various datasets about planets. For this reason, we automatically discover all (minimal) FDs. This introduces a new challenge, because we now deal with much larger, often spurious, but complete sets of FDs.

Using all FDs of a particular relational instance in the normalization process further introduces the challenge of selecting appropriate keys and foreign keys from the FDs (see Step (5)), because most of the FDs are coincidental, i.e., they are syntactically true but semantically false. This means that when the data changes these semantically invalid FDs could be violated and, hence, no longer work as a constraint. So we introduce features to automatically identify (and choose) reliable constraints from the set of FDs, which is usually too large for a human to manually examine.

Even if all FDs are semantically correct, selecting appropriate keys and foreign keys is still difficult. The decisions made here define which decompositions are executed, because decomposition options are often mutually exclusive: If, for instance, two violating FDs overlap, one split can make the other split infeasible. This happens, because BCNF normalization is not dependency preserving [12]. In all these constellations, however, some violating FDs are semantically better choices than others, which is why violating FDs must not only be filtered but also ranked by such quality features.

Another challenge, besides guiding the normalization process in the right direction, is the computational complexity of the normalization. Beeri and Bernstein have proven that the question “Given a set of FDs and a relational schema that embodies it, does the schema violate BCNF?” is NP-complete in the number of attributes [3]. To test this, we need to check that the Lhs of each of these FDs is a key or a super key, i.e., if each Lhs determines all other attributes. This is trivial if all FDs are transitively fully extended, i.e., they are transitively closed. For this reason, the complexity lies in calculating these closures (see Step (2)). Because no current algorithm is able to solve the closure calculation efficiently, we propose novel techniques for this sub-task of schema normalization.

Overall, the number of functional dependencies in datasets is typically much greater than a human expert can manually cope with [18]. A normalization algorithm must, therefore, be able to handle such very large inputs automatically.

Contributions. We propose a novel, instance-based schema normalization algorithm called NORMALIZE that can perform the normalization of a relational dataset automatically or supervised by an expert. Being able to put a human in the loop enables the algorithm to combine its analytical strengths with the domain knowledge of an expert. With NORMALIZE and this paper, we make the following contributions:

a) *Schema normalization.* We show how the entire schema normalization process can be implemented as one algorithm, which no previous work has done before. We discuss each component of this algorithm in detail. The main contribution of our (semi-)automatic approach is to incrementally weed out semantically false FDs by focusing on those FDs that are most likely true.

b) *Closure calculation.* We present two efficient closure algorithms, one for general FD result sets and one for complete result sets. Their core innovations include a more focused extension procedure, the use of efficient index-structures, and parallelization. These algorithms are not only useful in the normalization context, but also for many other FD-related tasks, such as query optimization, data cleansing, or schema reverse-engineering.

c) *Violation detection.* We propose a compact data structure, i.e., a prefix tree, to efficiently detect FDs that violate BCNF. This is the first approach to algorithmically improve this step. We also discuss how this step can be changed to discover violating FDs for normal forms other than BCNF.

d) *Constraint selection.* We contribute several features to rate the probability of key and foreign key candidates for actually being constraints. With the results, the candidates can be ranked, filtered, and selected as constraints during the normalization process. The selection can be done by either an expert or by the algorithm itself. Because all previous works on schema normalization assumed all input FDs to be correct, this is the first solution for a problem that has been ignored until now.

e) *Evaluation.* We evaluate our algorithms on several datasets demonstrating the efficiency of the closure calculation on complete, real-world FD result sets and the feasibility of (semi-)automatic schema normalization.

The remainder of this paper is structured as follows: First, we discuss related work in Section 2. Then, we introduce the schema normalization algorithm NORMALIZE in Section 3. The following sections go into more detail explaining the closure calculation in Section 4, the key derivation in Section 5, and the violation detection in Section 6. Section 7, then, introduces assessment techniques for key and foreign key candidates. The normalization algorithm is finally evaluated in Section 8 and we conclude in Section 9.

2. RELATED WORK

Normal forms for relational data have been extensively studied since the proposal of the relational data model itself [6]. For this reason, many normal forms have been proposed. Instead of giving a survey on normal forms here, we refer the interested reader to [10]. The Boyce-Codd Normal Form (BCNF) [7] is the most popular normal form, because it removes most kinds of redundancy from relational schemata. This is why we focus on this particular normal form in this paper. Most of the proposed techniques can, however, likewise be used to create other normal forms. The idea for our normalization algorithm follows the BCNF decomposition algorithm proposed in [12] and many other text books on database systems. The algorithm eliminates all anomalies related to functional dependencies while still guaranteeing full information recoverability via natural joins.

Schema normalization and especially the normalization into BCNF are well studied problems [3, 5, 16]. Bernstein presents a complete procedure for performing schema synthesis based on functional dependencies [4]. In particular, he shows that calculating the closure over a set of FDs is a crucial step in the normalization process. He also lays the theoretical foundation for our paper. But like most other works on schema normalization, Bernstein takes the functional dependencies and their semantic validity as a given – an assumption that hardly applies, because FDs are usually hidden in the data and must be discovered. For this reason, existing works on schema normalization greatly underestimate the number of valid FDs in non-normalized datasets and they also ignore the task of filtering the syntactically correct FDs for semantically meaningful ones. These reasons make those normalization approaches inapplicable in practice. In this paper, we propose a normalization system that covers the entire process from FD discovery over constraint selection up to the final relation decomposition. We show the feasibility of this approach in practical experiments.

There are other works on schema normalization, such as the work of Diederich and Milton [9], who understood that calculating the transitive closure over the FDs is a computational complex task that becomes infeasible facing real-world FD sets. As a solution, they propose to remove so called *extraneous* attributes from the FDs before calculating the closure, which reduces the calculation costs significantly. However, if all FDs are minimal, which is the case in our normalization process, then no *extraneous* attributes exist, and the proposed pruning strategy is futile.

One important difference between traditional normalization approaches and our algorithm is that we retrieve *all minimal FDs* from a given relational instance to exploit them for closure calculation (syntactic step) and constraint selection (semantic step). The latter has received little attention in previous research. In [2], Andritsos et al. proposed to rank the FDs used for normalization by the entropy of their attribute sets: The more duplication an FD removes, the better it is. The problem with this approach is that it weights the FDs only for effectiveness and not for semantic relevance. Entropy is also expensive to calculate, which is why we use different features. In fact, we use techniques inspired by [20], who extracted foreign keys from inclusion dependencies.

Schema normalization is a sub-task in schema design and evolution. There are numerous database administration tools, such as Navicat[1], Toad[2], and MySQL Workbench[3], that support these overall tasks. Most of them transform a given schema into an ER-diagram that a user can manipulate. All manipulations are then translated back to the schema and its data. Such tools are partly able to support normalization processes, but none of them can automatically propose normalizations with FDs retrieved from the data.

[1] https://www.navicat.com/

[2] http://www.toadworld.com/

[3] http://www.mysql.com/products/workbench/

In [3], the authors propose an efficient algorithm for the membership problem, i.e., the problem of testing whether one given FD is in the cover or not. This algorithm does not solve the closure calculation problem, but the authors propose some improvements in that algorithm that our improved closure algorithm uses as well, e.g., testing only for missing attributes on the RHS. They also propose derivation trees as a model for FD derivations, i.e., deriving further FDs from a set of known FDs using Armstrong's inference rules. Because no algorithm is given for their model, we cannot compare our solution against it.

As stated above, the discovery of functional dependencies from relational data is a prerequisite for schema normalization. Fortunately, FD discovery is a well researched problem and we find various algorithms to solve it. In this work, we utilize the HYFD algorithm, which is the most efficient FD discovery algorithm at the time [19]. This algorithm discovers – like almost all FD discovery algorithms – the complete set of all minimal, syntactically valid FDs in a given relational dataset. We exploit these properties, i.e., minimality and completeness in our closure algorithm.

3. SCHEMA NORMALIZATION

To normalize a schema into Boyce-Codd Normal Form (BCNF), we implement the straightforward BCNF decomposition algorithm shown in most textbooks on database systems, such as [12]. The BCNF-conform schema produced by this algorithm is always a tree-shaped *snowflake schema*, i.e., the foreign key structure is hierarchical and cycle-free. For this reason, our normalization algorithm is not designed to (re-)construct arbitrary non-snowflake schemata. It, however, removes all redundancy related to functional dependencies from the relations. If other schema design decisions that lead to alternative schema topologies are necessary, the user must (and can!) interactively choose different decompositions other than the ones our algorithm can propose.

In the following, we propose a normalization process that takes an arbitrary relational instance as input and returns a BCNF-conform schema for it. The input dataset can contain one or more relations, and no other metadata than the dataset's schema is required. This schema, which is incrementally changed during the normalization process, is globally known to all algorithmic components. We refer to a dataset's *schema* as its set of relations, specifying attributes, tables, and key/foreign key constraints. For instance, the schema of our example dataset in Table 2 is $\{R_1(\underline{\textsf{First}}, \underline{\textsf{Last}}, \textsf{Postcode}), R_2(\underline{\textsf{Postcode}}, \textsf{City}, \textsf{Mayor})\}$. Underlined attributes represent keys and same attribute names represent foreign keys.

Figure 1 gives an overview of the normalization algorithm, which we call NORMALIZE. In contrast to other normalization algorithms, such as those proposed in [4] or [9], NORMALIZE does not have any components responsible for minimizing FDs or removing extraneous FDs. This is because the set of FDs on which we operate, is not arbitrary; it contains only minimal and, hence, no extraneous FDs due to the FD discovery step. We now introduce the components step by step and discuss the entire normalization process.

(1) FD Discovery. Given a relational dataset, the first component is responsible for discovering all minimal functional dependencies. Any known FD discovery algorithm, such as TANE [14] or DFD [1], can be used, because all these algorithms are able to discover the complete set of minimal FDs in relational datasets. We make use of our HYFD [19] algorithm here, because it is the most efficient algorithm for this task and it offers special pruning capabilities that we can exploit later in the normalization process. In summary, the first component reads the data, discovers all FDs, and sends them to the second component. For more details on this discovery step, we refer to [19].

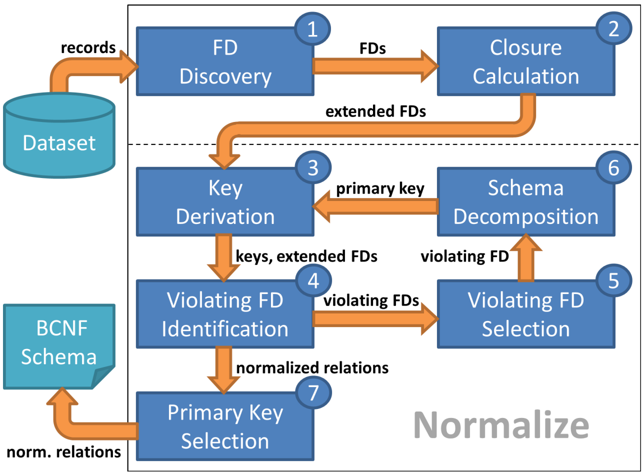


Figure 1: "Normalize" and its components.

(2) Closure Calculation. The second component calculates the closure over the given FDs. The closure is needed by subsequent components to infer keys and normal form violations. Formally, the *closure* X_F^+ *over a set of attributes* X given the FDs F is defined as the set of attributes X plus all additional attributes Y that we can add to X using F and Armstrong's transitivity axiom [9]. If, for example, $X = \{A, B\}$ and $F = \{A \rightarrow C, C \rightarrow D\}$, then $X_F^+ = \{A, B, C, D\}$. We now define the *closure* F^+ *over a set of FDs* F as a set of extended FDs: The RHS Y of each FD $X \rightarrow Y \in F$ is extended such that $X \cup Y = X_F^+$. In other words, each FD in F is maximized using Armstrong's transitivity axiom. Because, as Beeri et al. have shown [3], this is an NP-hard task with respect to the number of attributes in the input relation, we shall propose an efficient FD extension algorithm that finds transitive dependencies via prefix tree lookups. This algorithm iterates the set of FDs only once and is able to parallelize its work. It exploits the fact that the given FDs are minimal and complete (Section 4).

(3) Key Derivation. The key derivation component collects those keys from the extended FDs that the algorithm requires for schema normalization. Such a key X is a set of attributes for which $X \rightarrow Y \in F^+$ and $X \cup Y = R_i$ with R_i being *all* attributes of relation i. In other words, if X determines all other attributes, it is a key for its relation. Once discovered, these keys are passed to the next component. Our method of deriving keys from extended functional dependencies does not reveal all existing keys in the schema, but we prove in Section 5 that only the derived keys are needed for BCNF normalization.

(4) Violating FD Identification. Given the extended FDs and the set of keys, the violation detection component checks all relations for being BCNF-conform. Recall that a relation R is BCNF-conform, iff for all FDs $X \rightarrow A$ in that relation the LHS X is either a key or superkey. So NORMALIZE checks the LHS of each FD for having a (sub)set in the set of keys; if no such (sub)set can be found, the FD is reported as a BCNF violation. Note that one could setup other normalization criteria in this component to accomplish 3NF or other normal forms. If FD violations were identified, these are reported to the next component; otherwise, the schema is BCNF-conform and can be sent to the primary key selection. We propose an efficient technique to find all violating FDs in Section 6.

(5) Violating FD Selection. The violating FD selection component is called with a set of violating FDs, if some relations are not yet in BCNF. In this case, the component scores all violating FDs for being good foreign key constraints. With these scores, the algorithm creates a ranking of violating FDs for each non-BCNF relation. From each ranking, a user picks the most suitable violating FD for normalization; if no user is present, the algorithm automatically picks the top ranked FD. Note that the user, if present, can also decide to pick none of the FDs, which ends the normalization process for the current relation. This is reasonable if all presented FDs are obviously semantically incorrect, i.e., the FDs hold on the given data accidentally but have no real meaning. Such FDs are presented with a relatively low score at the end of the ranking. Eventually, the iterative process automatically weeds out most of the semantically incorrect FDs by selecting only semantically reliable FDs in each step. We discuss the violating FD selection together with the key selection in Section 7.

(6) Schema Decomposition. Knowing the violating FDs, the actual schema decomposition is a straight-forward task: Each relation R, for which a violating FD $X \to Y$ is given, is split into two parts – one part without the redundant attributes $R_1 = R \backslash Y$ and one part with the FD's attributes $R_2 = X \cup Y$. Now X automatically becomes the new primary key in R_2 and a foreign key in R_1. With these new relations, the algorithm goes back into step (3), the key selection, because new keys might have appeared in R_2, namely those keys Z for which $Z \to X$ holds. Because the decomposition itself is straightforward, we do not go into more detail for this component in this paper.

(7) Primary Key Selection. The primary key selection is the last component in the normalization process. It makes sure that every BCNF-conform relation has a primary key constraint. Because the decomposition component already assigns keys and foreign keys when splitting relations, most relations already have a primary key. Only those relations that had no primary key at the beginning of the normalization process are processed by this component. For them, the algorithm assigns a primary key in a (semi-)automated way: All keys of the respective relation are scored for being a good primary key; then the keys are ranked by their score and either a human picks a primary key from this ranking, or the algorithm automatically picks the highest ranked key as the relation's primary key. Section 7 describes the scoring and selection of keys in more detail.

Once the closure of all FDs is calculated, the components (3) to (6) form a loop: This loop drives the normalization process until component (4) finds the schema to be BCNF-conform. Overall, the proposed components can be grouped into two classes: The first class includes the components (1), (2), (3), (4), and (6) and operates on a syntactic level; the results in this class are well defined and the focus is set on performance optimization. The second class includes the components (5) and (7) and operates on a semantic level; the computations here are easy to execute but the choices are difficult and the quality of the result matters.

4. CLOSURE CALCULATION

Armstrong formulated the following three axioms for functional dependencies on attribute sets X, Y, and Z [3]:

1. *Reflexivity*: If $Y \subseteq X$, then $X \to Y$.
2. *Augmentation*: If $X \to Y$, then $X \cup Z \to Y \cup Z$.
3. *Transitivity*: If $X \to Y$ and $Y \to Z$, then $X \to Z$.

For schema normalization, we are given a set of FDs F and need to find a cover F^+ that maximizes the right hand side of each FD in F. The maximization of FDs is important to identify keys and to decompose relations correctly. In our running example, for instance, we might be given Postcode→City and City→Mayor. A correct decomposition with foreign key Postcode requires Postcode→City,Mayor; otherwise we would lose City→Mayor, because the attributes City and Mayor would end up in different relations. Therefore, we apply Armstrong's transitivity axiom on F to calculate its cover F^+.

The closure F^+ extends each FD using Armstrong's reflexivity and transitivity axioms. Augmentation need not be used, because this rule generates new, non-minimal FDs instead of extending existing ones. The decomposition steps require the FDs' LHS to be minimal, i.e., removing any attribute from X would invalidate $X \to Y$, because X should become a *minimal* key after decomposition.

The reflexivity axiom adds all LHS attributes to an FD's RHS. To reduce memory consumption, we make this extension only implicit: We assume that LHS attributes always also belong to an FD's RHS without explicitly storing them on that side. For this reason, we apply the transitivity axiom for attribute sets W, X, Y, and Z as follows: If $W \to X$, $Y \to Z$, and $Y \subseteq W \cup X$, then $W \to Z$. So if, for instance, the FD First,Last→Mayor is given, we can extend the FD First,Postcode→Last with the RHS attribute Mayor, because {First, Last} $\subseteq$ {First, Postcode} $\cup$ {Last}.

In the following, we discuss three algorithms for calculating F^+ from F: A naive algorithm, an improved algorithm for arbitrary sets of FDs, and an optimized algorithm for complete sets of minimal FDs. While the second algorithm might be useful for closure calculation in other contexts, such as query optimization or data cleansing, we recommend the third algorithm for our normalization system. All three algorithms store F, which is transformed into F^+, in the variable *fds*.

4.1 Naive closure algorithm

The naive closure algorithm, which was already introduced as such in [9], is given as Algorithm 1. For each functional dependency in *fds* (Line 3), the algorithm iterates all other FDs (Line 4) and tests if these extend the current FD (Line 5). If an extension is possible, the current FD is updated (Line 6). These updates might enable further updates for already tested FDs. For this reason, the naive algorithm iterates the FDs until an entire pass has not added any further extensions (Line 8).

Algorithm 1: Naive Closure Calculation

```
  Data: fds
  Result: fds
1 do
2     somethingChanged ← false;
3     foreach fd ∈ fds do
4         foreach otherFd ∈ fds do
5             if otherFd.lhs ⊆ fd.lhs ∪ fd.rhs then
6                 fd.rhs ← fd.rhs ∪ otherFd.rhs;
7                 somethingChanged ← true;
8 while somethingChanged ;
9 return fds;
```

4.2 Improved closure algorithm

There are several ways to improve the naive closure algorithm, some of which have already been proposed in similar form in [9] and [3]. We now present an improved closure algorithm that solves the following three issues: First, the algorithm should not check all other FDs when extending one specific FD, but only those that can possibly link to a missing Rhs attribute. Second, when looking for a missing Rhs attribute, the algorithm should not check all other FDs that can provide it, but only those that have a subset-relation with the current FD, i.e., those that are relevant for extensions. Third, the change-loop should not iterate the entire FD set, because some FDs must be extended more often than others so that many extension tests are executed superfluously.

Algorithm 2 shows our improved version. First, we remove the nested loop over all other FDs and replace it with index lookups. The index structure we propose is a set of prefix-trees, aka. tries. Each trie stores all FD Lhss that have the same, trie-specific Rhs attribute. Having an index for each Rhs attribute allows the algorithm to check only those other FDs that can deliver a link to a Rhs attribute that a current FD is actually missing (Line 8).

The *lhsTries* are constructed before the algorithm starts extending the given FDs (Lines 1 to 4). Each index-lookup must then not iterate all FDs referencing the missing Rhs attribute; it instead performs a subset search in the according prefix tree, because the algorithm is specifically looking for an FD whose Lhs is contained in the current FD's Rhs attributes (Line 9). The subset search is much more effective than iterating all possible extension candidates and has already been proposed for FD generalization lookups in [11].

As a third optimization, we propose to move the change-loop inside the FD-loop (Line 6). Now, a single FD that requires many transitive extensions in subsequent iterations does not trigger the same number of iterations over all FDs, which mostly are already fully extended.

4.3 Optimized closure algorithm

Algorithm 2 works well for all sets of FDs, but we can further optimize the algorithm with the assumption that these sets contain *all minimal FDs*. Algorithm 3 shows this more efficient version for complete sets of minimal FDs.

Like Algorithm 2, the optimized closure algorithm also uses the Lhs tries for efficient FD extensions, but it does not require a change-loop so that it iterates the missing Rhs attributes of an FD only once. The algorithm also checks only the Lhs attributes of an FD for subsets and not all attributes of a current FD (Line 7). These two optimizations are possible, because the set of FDs is complete and minimal so that we always find a subset-FD for any valid extension attribute. The following lemma states this formally:

Algorithm 2: Improved Closure Calculation

```
   Data: fds
   Result: fds
 1 array lhsTries size | schema.attributes | as trie;
 2 foreach fd ∈ fds do
 3     foreach rhsAttr ∈ fd.rhs do
 4         lhsTries[rhsAttr].insert (fd.lhs);
 5 foreach fd ∈ fds do
 6     do
 7         somethingChanged ← false;
 8         foreach attr ∉ fd.rhs ∪ fd.lhs do
 9             if fd.lhs ∪ fd.rhs ⊇ lhsTries[attr] then
10                 fd.rhs ← fd.rhs ∪ attr;
11                 somethingChanged ← true;
12     while somethingChanged ;
13 return fds;
```

Lemma 1. *Let F be a complete set of minimal FDs. If $X \rightarrow Y \in F$ and $X \rightarrow A$ with $A \notin Y$ is valid, then there must exist an $X' \subset X$ so that $X' \rightarrow A \in F$.*

Proof. If $X \rightarrow A$ and $X \rightarrow A \notin F$, then $X \rightarrow A$ is not minimal and a minimal FD $X' \rightarrow A$ with $X' \subset X$ must exist. If $X' \rightarrow A \notin F$, then F is not a complete set of minimal FDs, which contradicts the premise that F is complete. □

The fact that all minimal FDs are required for Algorithm 3 to work correctly has the disadvantage that complete sets of FDs are usually much larger than sets of FDs that have already been reduced to meaningful FDs. Reducing a set of FDs to meaningful ones is, on the contrary, a difficult and use-case specific task that becomes more accurate if the FDs' closure is known. For this reason, we perform the closure calculation before the FD selection and accept the increased processing time and memory consumption.

The increased processing time is hardly an issue, because the performance gain of Algorithm 3 over Algorithm 2 on same sized inputs is so significant that larger sets of FDs can still easily be processed. We show this in Section 8. The

Algorithm 3: Optimized Closure Calculation

```
  Data: fds
  Result: fds
1 array lhsTries size | schema.attributes | as trie;
2 foreach fd ∈ fds do
3     foreach rhsAttr ∈ fd.rhs do
4         lhsTries[rhsAttr].insert (fd.lhs);
5 foreach fd ∈ fds do
6     foreach attr ∉ fd.rhs ∪ fd.lhs do
7         if fd.lhs ⊇ lhsTries[attr] then
8             fd.rhs ← fd.rhs ∪ attr;
9 return fds;
```

increased memory consumption, on the other hand, becomes a problem if the complete set of minimal FDs is too large to be held in memory or maybe even too large to be held on disk. We then need to prune FDs, but which FDs can be pruned so that Algorithm 3 still computes a correct closure on the remainder? To fully extend an FD $X \rightarrow Y$, the algorithm requires all subset-FDs $X' \rightarrow Z$ with $X' \subset X$ to be available. So if we prune all superset-FDs with larger LHS than $|X|$, the calculated closure for $X \rightarrow Y$ and all its subset-FDs $X' \rightarrow Z$ would still be correct. In general, we can define a maximum LHS size and prune all FDs with a larger LHS size while still being able to compute the complete and correct closure for the remaining FDs with Algorithm 3. This pruning fits our normalization use-case well, because FDs with shorter LHS are semantically better candidates for key and foreign key constraints as we argue in Section 7. NORMALIZE achieves the maximum LHS size pruning for free, because it is already implemented in the HYFD algorithm that we proposed using for the FD discovery.

All three closure algorithms can easily be parallelized by splitting the FD-loops (Lines 3, 2, and 5 respectively) to different worker threads. This is possible, because each worker changes only its own FD and changes made to other FDs can, but do not have to be seen by this worker.

Considering the complexity of the three algorithms with respect to the number of input FDs, the naive algorithm is in $\mathcal{O}(|fds|^3)$, the improved in $\mathcal{O}(|fds|^2)$ and the optimized in $\mathcal{O}(|fds|)$. But because the number of FDs potentially increases exponentially with the number of attributes, all three algorithms are NP-complete in the number of attributes. We compare the algorithms experimentally in Section 8.

5. KEY DERIVATION

Keys are important in normalization processes, because they do not contain any redundancy due to their uniqueness. Hence, they do not cause anomalies in the data. Keys basically indicate normalized schema elements that do not need to be decomposed, i.e., decomposing them would not remove any redundancy in the given relational instance. In this section, we first discuss how keys can be derived from extended FDs. Then, we prove that the set of derived keys is sufficient for BCNF schema normalization.

Deriving keys from extended FDs. By definition, a key is any attribute or attribute combination whose values uniquely define all other records [6]. In other words, the attributes of a key X functionally determine all other attributes Y of a relation R. So given the extended FDs, the keys can easily be found by checking each FD $X \rightarrow Y$ for $X \cup Y = R$.

The set of keys that we can directly derive from the extended FDs does, however, not necessarily contain *all* minimal keys of a given relation. Consider here, for instance, the relations $\mathsf{Professor}(\underline{\mathsf{name}}, \mathsf{department}, \mathsf{salary})$, $\mathsf{Teaches}(\underline{\mathsf{name}}, \underline{\mathsf{label}})$, and $\mathsf{Class}(\underline{\mathsf{label}}, \mathsf{room}, \mathsf{date})$ with Teaches being a join table for the n:m-relationship between Professor and Class. When we *denormalize* this schema by calculating $R = \mathsf{Professor} \bowtie \mathsf{Teaches} \bowtie \mathsf{Class}$, we get $R(\underline{\mathsf{name}}, \underline{\mathsf{label}}, \mathsf{department}, \mathsf{salary}, \mathsf{room}, \mathsf{date})$ with primary key $\{\mathsf{name}, \mathsf{label}\}$. This key *cannot* directly be derived from the minimal FDs, because $\mathsf{name{,}label}\rightarrow A$ is not a minimal FD for any $A \in R_i$; the two minimal FDs are $\mathsf{name}\rightarrow\mathsf{department{,}salary}$ and $\mathsf{label}\rightarrow\mathsf{room{,}date}$.

Skipping missing keys. The discovery of missing keys is an expensive task, especially when we consider the number of FDs that can be huge for non-normalized datasets. The BCNF-normalization, however, only requires those keys that we can directly derive from the extended FDs. We can basically ignore the missing keys, because the algorithm checks normal form violations only with keys that are *subsets* of an FD's LHS (see Section 6) and all such keys can directly be derived. The following lemma states this more formally:

LEMMA 2. *If X' is a key and $X \rightarrow Y \in F^+$ is an FD with $X' \subseteq X$, then X' can directly be derived from F^+.*

PROOF. Let X' be a key of relation R and let $X \rightarrow Y \in F^+$ be an FD with $X' \subseteq X$. To directly derive the key X' from F^+, we must prove the existence of an FD $X' \rightarrow Z \in F^+$ with $Z = R \setminus X'$.

X must be a minimal LHS in some FD $X \rightarrow Y'$ with $Y' \subseteq Y$, because $X \rightarrow Y \in F^+$ and F is the set of all minimal FDs. Now consider the precondition $X' \subseteq X$: If $X' \subset X$, then $X \rightarrow Y \notin F^+$, because X is a key and, hence, it determines any attribute A that X could contain more than X'. Therefore, $X = X'$ must be true. At this point, we have that $X \rightarrow Y' \in F^+$ and $X = X'$. So $X' \rightarrow Y' \in F^+$ must be true as well, which also shows that $Y' = Y = Z$, because X' is a key. □

The key derivation component in NORMALIZE in fact discovers only those keys that are relevant for the normalization process by checking $X \cup Y = R$ for each FD $X \rightarrow Y$. The primary key selection component in the end of the normalization process must, however, discover all keys for those relations that did not receive a primary key from any previous decomposition operation. For this task, we use the DUCC algorithm by Heise et al. [13], which is specialized in key discovery. The key discovery is an NP complete problem, but because the normalized relations are much smaller than the non-normalized starting relations, it is a fast operation at this stage of the algorithm.

6. VIOLATION DETECTION

Given the extended *fds* and the *keys*, detecting BCNF violations is straightforward: Each FD whose LHS is neither a key nor a super-key must be classified as a violation. Algorithm 4 shows how this can be efficiently done again using a prefix tree for subset searches.

At first, the violation detection algorithm inserts all given keys into a trie (Lines 1 to 3). Then, it iterates the *fds* and, for each FD, it checks if the FD's LHS contains a `null` value $\perp$. Such FDs do not need to be considered for decompositions, because the LHS becomes a primary key constraint in the new, split off relation and SQL prohibits `null` values in key constraints. Note that there is work on *possible/certain key constraints* that permit $\perp$ values in keys [15], but we continue with the standard for now. If the LHS contains no `null` values, the algorithm queries the *keyTrie* for subsets of the FD's LHS (Line 8). If a subset is found, the FD does not violate BCNF and we continue with the next FD; otherwise, the FD violates BCNF.

To preserve existing constraints, we remove all primary key attributes from a violating FD's RHS, if a primary key is present (Line 11). Not removing the primary key attributes from the FD's RHS could cause the decomposition step to break the primary key apart. Some key attributes would

Algorithm 4: Violation Detection

```
   Data: fds, keys
   Result: violatingFds
1  keyTrie ← new trie;
2  foreach key ∈ keys do
3      keyTrie.insert(key);
4  violatingFds ← ∅;
5  foreach fd ∈ fds do
6      if ⊥ ∈ fd.lhs then
7          continue;
8      if fd.lhs ⊇ keyTrie then
9          continue;
10     if currentSchema.primaryKey ≠ null then
11         fd.rhs ← fd.rhs − currentSchema.primaryKey;
12     if ∃ fk ∈ currentSchema.foreignKeys:
13         (fk ∩ fd.rhs ≠ ∅) ∧ (fk ⊈ fd.lhs ∪ fd.rhs) then
14         continue;
15     violatingFds ← violatingFds ∪ fd;
16 return violatingFds;
```

then be moved into another relation breaking the primary key constraint and possible foreign key constraints referencing this primary key. Because the current schema might also contain foreign key constraints, we test if the violating FD preserves all such constraints when used for decomposition: Each foreign key *fk* must stay intact in either of the two new relations or otherwise we do not use the violating FD for normalization (Line 12). The algorithm finally adds each constraint preserving violating FD to the *violatingFds* result set (Line 15). In Section 7 we propose a method to select one of them for decomposition.

When a violating FD $X \rightarrow Y$ is used to decompose a relation R, we obtain two new relations, which are $R_1(R \backslash Y \cup X)$ and $R_2(X \cup Y)$. Due to this split of attributes, not all previous FDs hold in R_1 and R_2. It is obvious that the FDs in R_1 are exactly those FDs $V \rightarrow W$ for which $V \cup W \subseteq R_1$ and $V \rightarrow W' \in F^+$ with $W \subseteq W'$, because the records for $V \rightarrow W$ are still the same in R_1; R_1 just lost some attributes that are irrelevant for all $V \rightarrow W$. The same observation holds for R_2 although the number of records has been reduced:

LEMMA 3. *The relation $R_2(X \cup Y)$ produced by a decomposition on FD $X \rightarrow Y$ retains exactly all FDs $V \rightarrow W$, for which $V \cup W \subseteq R_2$ and $V \rightarrow W$ is valid in R.*

PROOF. (1) Any valid $V \rightarrow W$ of R is still valid in R_2: Assume that $V \rightarrow W$ is valid in R but invalid in R_2. Then R_2 must contain at least two records violating $V \rightarrow W$. Because the decomposition only *removes* records in $V \cup W$ and $V \cup W \subseteq R_2 \subseteq R$, these violating records must also exist in R. But such records cannot exist in R, because $V \rightarrow W$ is valid in R; hence, the FD must also be valid in R_2.

(2) No valid $V \rightarrow W$ of R_2 can be invalid in R: Assume $V \rightarrow W$ is valid in R_2 but invalid in R. Then R must contain at least two records violating $V \rightarrow W$. Because these two records are not completely equal in their $V \cup W$ values and $V \cup W \subseteq R_2$, the decomposition does not remove them and they also exist in R_2. So $V \rightarrow W$ must also be invalid in R_2. Therefore, there can be no FD valid in R_2 but invalid in R. □

Assume that, instead of BCNF, we would aim to assure 3NF, which is slightly less strict than BCNF: In contrast to BCNF, 3NF does not remove all FD-related redundancy, but it is dependency preserving. Consequently, no decomposition may split an FD other than the violating FD [4]. To calculate 3NF instead of BCNF, we could additionally remove all those groups of violating FDs from the result of Algorithm 4 that are mutually exclusive, i.e., any FD that would split the LHS of some other FD. To calculate stricter normal forms than BCNF, we would need to have detected other kinds of dependencies. For example, constructing 4NF requires all multi-valued dependencies (MVDs) and, hence, an algorithm that discovers MVDs. The normalization algorithm, then, would work in the same manner.

7. CONSTRAINT SELECTION

During schema normalization, we need to define key and foreign key constraints. Syntactically, all keys are equally correct and all violating FDs form correct foreign keys, but semantically the choice of primary keys and violating FDs makes a difference. Judging the relevance of keys and FDs from a semantic point of view is a difficult task for an algorithm – and in many cases for humans as well – but in the following, we define some quality features that serve to automatically score keys and FDs for being "good" constraints, i.e., constraints that are not only valid on the given instance but are true for its schema.

The two selection components of NORMALIZE use these features to score the key and foreign-key candidates, respectively. Then, they sort the candidates by their score. The most reasonable candidates are presented at the top of the list and likely accidental candidates appear at the end. By default, NORMALIZE uses the top-ranked candidate and proceeds; if a user is involved, she can choose the constraint or stop the process. The candidate list can, of course, become too large for a full manual inspection, but (1) the user always needs to pick only one element, i.e., she does not need to classify all elements in the list as either true or false, (2) the candidate list becomes shorter in every step of the algorithm as many options are implicitly weeded out, and (3) the problem of finding a split candidate in a ranked enumeration of options is easier than finding a split without any ordering, as it would be the case without our method.

7.1 Primary key selection

If a relation has no primary key, we must assign one from the relation's set of keys. To find the semantically best key, NORMALIZE scores all keys X using the following features:

(1) Length score: $\frac{1}{|X|}$

Semantically correct keys are usually shorter than random keys (in their number of attributes $|X|$), because schema designers tend to use short keys: Short keys can more efficiently be indexed and they are easier to understand.

(2) Value score: $\frac{1}{max(1, |max(X)| - 7)}$

The values in primary keys are typically short, because they serve to identify records and usually do not contain much business logic. Most relational database management systems (RDBMS) also restrict the maximum length of values in primary key attributes, because primary keys are indexed by default and indices with too long values are more difficult to manage. So we downgrade keys with values longer than 8 characters using the function *max(X)* that returns the longest value in attribute (combination) X; for multiple attributes, *max(X)* concatenates their values.

(3) Position score: $\frac{1}{2}(\frac{1}{|left(X)|+1} + \frac{1}{|between(X)|+1})$
When considering the order of attributes in their relations, key attributes are typically located left and without non-key attributes between them. This is intuitive, because humans tend to place keys first and logically coherent attributes together. The position score exploits this by assigning decreasing score values to keys depending on the number of non-key attributes left *left(X)* and between *between(X)* key attributes X.

The formulas we propose for the ranking reflect only our intuition. The list of features is most likely also not complete, but the proposed features produce good results for key scoring in our experiments. For the final *key score*, we simply calculate the mean of the individual scores. The perfect key with one attribute, a maximum value length of 8 characters and position one in the relation, then, has a key score of 1; less perfect keys have lower scores.

After scoring, Normalize ranks the keys by their score and lets the user choose a primary key amongst the top ranked keys; if no user interaction is desired (or possible), the algorithm automatically selects the top-ranked key.

7.2 Violating FD selection

During normalization, we need to select some violating FDs for the schema decompositions. Because the selected FDs become foreign key constraints after the decompositions, the violating FD selection problem is similar to the foreign key selection problem [20], which scores inclusion dependencies (INDs) for being good foreign keys. The viewpoints are, however, different: Selecting foreign keys from INDs aims to identify semantically correct links between existing tables; selecting foreign keys from FDs, on the other hand, is about forming redundancy-free tables with appropriate keys.

Recall that selecting semantically correct violating FDs is crucial, because some decompositions are mutually exclusive. If possible, a user should also discard violating FDs that hold only accidentally in the given relational instance. Otherwise, Normalize might drive the normalization a bit too far by splitting attribute sets – in particular sparsely populated attributes – into separate relations.

In the following, we discuss our features for scoring violating FDs $X \rightarrow Y$ as good foreign key constraints:

(1) Length score: $\frac{1}{2}(\frac{1}{|X|} + \frac{1}{|Y| \cdot (|R|-2)})$
Because the Lhs X of a violating FD becomes a primary key for the Lhs attributes after decomposition, it should be short in length. The Rhs Y, on the contrary, should be long so that we create large new relations: Large right-hand sides not only raise the confidence of the FD to be semantically correct, they also make the decomposition more effective. Because the Rhs can be at most $|R| - 2$ attributes long in relation R (one attribute must be X and one must not depend on X so that X is not a key in R), we weight the Rhs's length by this factor.

(2) Value score: $\frac{1}{max(1,|max(X)|-7)}$
The value score for a violating FD is the same as the value score for a primary key X, because X becomes a primary key after decomposition.

(3) Position score: $\frac{1}{2}(\frac{1}{|between(X)|+1} + \frac{1}{|between(Y)|+1})$
The attributes of a semantically correct FD are most likely placed close to one another due to their common context. We expect this to hold for both the FD's Lhs and Rhs. The space between Lhs and Rhs attributes, however, is only a very weak indicator, and we ignore it. For this reason, we weight the violating FD anti-proportionally to the number of attributes *between* Lhs attributes and *between* Rhs attributes.

(4) Duplication score: $\frac{1}{2}(2 - \frac{|uniques(X)|}{|values(X)|} - \frac{|uniques(Y)|}{|values(Y)|})$
A violating FD is well suited for normalization if both Lhs X and Rhs Y contain possibly many duplicate values and, hence, much redundancy. The decomposition can, then, remove many of these redundant values. As for most scoring features, a high duplication score in the Lhs values reduces the probability that the FD holds by coincidence, because only duplicate values in an FD's Lhs can invalidate the FD and having many duplicate values in Lhs X without any violation is a good indicator for its semantic correctness. For scoring, we estimate the number of unique values in X and Y with $|uniques()|$; because exactly calculating this number is computationally expensive, we create a Bloom-filter for each attribute and use their false positive probabilities to efficiently estimate the number of unique values.

We calculate the final *violating FD score* as the mean of the individual scores. In this way, the most promising violating FD is one that has a single Lhs attribute determining almost the entire relation with short and few distinct values. Like for the key scoring, the proposed features reflect our intuitions and observations; they might not be optimal or complete, but they produce reasonable results for a difficult selection problem: In our experiments the top-ranked violating FDs usually indicate the semantically best decomposition points.

After choosing a violating FD for becoming a foreign key constraint, we could in principle decide to remove indovidual attributes from the FD's Rhs. One reason might be that these attributes also appear in another FD's Rhs and can be used in a subsequent decomposition. So when a user guides the normalization process, we present all Rhs attributes that are also contained in other violating FDs. He/she can then decide to remove such attributes. If no user is present, nothing is removed.

8. EVALUATION

In this section, we evaluate the efficiency and effectiveness of our normalization algorithm Normalize. At first, we introduce our experimental setup. Then, we evaluate the performance of Normalize and in particular its closure calculation component. In the end, we assess the quality of the normalization output.

8.1 Experimental setup

Normalize has been implemented using the *Metanome* data profiling framework (www.metanome.de), which defines standard interfaces for different kinds of profiling algorithms [17]. In particular, Metanome provided the implementation of the HyFD FD discovery algorithm. Common tasks, such as input parsing, result formatting, and performance measurement, are standardized by the framework and decoupled from the algorithm itself.

Table 3: The datasets, their characteristics, and their processing times

Name	Size	Attr.	Records	FDs	FD-Keys	FD Disc.	Closure$_{impr}$	Closure$_{opt}$	Key Der.	Viol. Iden.
Horse	25.5 kB	27	368	128,727	40	4,157 ms	1,765 ms	486 ms	40 ms	246 ms
Plista	588.8 kB	63	1000	178,152	1	9,847 ms	6,652 ms	857 ms	49 ms	55 ms
Amalgam1	61.6 kB	87	50	450,020	2,737	3,462 ms	745 ms	333 ms	7 ms	25 ms
Flight	582.2 kB	109	1000	982,631	25,260	20,921 ms	132,085 ms	1,662 ms	77 ms	93 ms
MusicBrainz	1.2 GB	106	1,000,000	12,358,548	0	2,132 min	215.5 min	1.4 min	331 ms	26 ms
TPC-H	6.7 GB	52	6,001,215	13,262,106	347,805	3,651 min	3.8 min	0.5 min	163 ms	4093 ms

Hardware. We ran all our experiments on a Dell PowerEdge R620 with two Intel Xeon E5-2650 2.00 GHz CPUs and 128 GB DDR3 RAM. The server runs on CentOS 6.7 and uses OpenJDK 64-Bit 1.8.0_71 as Java environment.

Datasets. We primarily use the synthetic *TPC-H*[4] dataset (scale factor one), which models generic business data, and the *MusicBrainz*[5] dataset, which is a user-maintained encyclopedia on music and artists. To evaluate the effectiveness of NORMALIZE, we denormalized the two datasets by joining all their relations into a single, universal relation. In this way, we can compare the normalization result to the original datasets. For MusicBrainz, we had to restrict this join to eleven selected core tables, because the number of tables in this dataset is huge. We also limited the number of records for the denormalized MusicBrainz dataset, because the associative tables produce an enormous amount of records when used for complete joins.

For the efficiency evaluation, we use four additional datasets, namely Horse, Plista, Amalgam1, and Flight. We provide these datasets and more detailed descriptions on our web-page[6]. In our evaluation, each dataset consists of one relation with the characteristics shown in Table 3; the input of NORMALIZE can, in general, consist of multiple relations.

8.2 Efficiency analysis

Table 3 lists six datasets with different properties. The amount of minimal functional dependencies in these datasets is between 128 thousand and 13 million, and thus too great to manually select meaningful ones. The column *FD-Keys* counts all those keys that we can directly derive from the FDs. Their number does not depend on the number of FDs but on the structure of the data: Amalgam1 and TPC-H have a snow-flake schema while, for instance, MusicBrainz has a more complex link structure in its schema.

We executed NORMALIZE on each of these datasets and measured the execution time for the components (1) FD Discovery, (2) Closure Calculation, (3) Key Derivation, and (4) Violating FD Identification. The first two components are parallelized so that they fully use all 32 cores of our evaluation machine. The necessary discovery of the complete FD set still requires 36 and 61 hours on the two larger datasets, respectively.

First of all, we notice that the key derivation and violating FD identification steps are much faster than the FD discovery and closure calculation steps; they usually finish in less than a second. This is important, because the two components are executed multiple times in the normalization process and a user might be in the loop interacting with the system at the same time. In Table 3, we show only the execution times for the first call of these components; subsequent calls can be handled even faster, because their input sizes shrink continuously. The time needed to determine the violating FDs depends primarily on the number of FD-keys, because the search for LHS generalizations in the trie of keys is the most expensive operation. This explains the long execution time of 4 seconds for the TPC-H dataset.

For the closure calculation, Table 3 shows the execution times of the improved (*impr*) and optimized (*opt*) algorithm. The naive algorithm already took 13 seconds for the Amalgam1 dataset (compared to less than 1 s for both *impr* and *opt*), 23 minutes for Horse (<2 s and <1 s for *impr* and *opt*, respectively), and 41 minutes for Plista (<7 s and <1 s). These runtimes are so much worse than the improved and optimized algorithm versions that we stopped testing it. The optimized closure algorithm, then, outperforms the improved version by factors of 2 (Amalgam1) to 159 (MusicBrainz), because it can exploit the completeness of the given FD set. The more extensions of right-hand sides the algorithm must perform, the higher this advantage becomes. The average RHS size for Amalgam1 FDs, for instance, increases from 32 to 56, whereas the average RHS size for MusicBrainz FDs increases from 3 to 40. For TPC-H, the average RHS size increases from 10 to 23. The runtimes of the optimized closure calculation are, overall, acceptable when compared to the FD discovery time. Therefore, it is not necessary to filter FDs prior to the closure calculation.

Because closure calculation is not only important for normalization but for many other use cases as well, Figure 2 analyses the scalability of this step in more detail. The graphs show the execution times of the improved and the optimized algorithm for an increasing number of input FDs. The experiment takes these input FDs randomly from the 12 million MusicBrainz FDs; the number of attributes is kept constant to 106. We again omit the naive algorithm, because it is orders of magnitude slower than both other approaches.

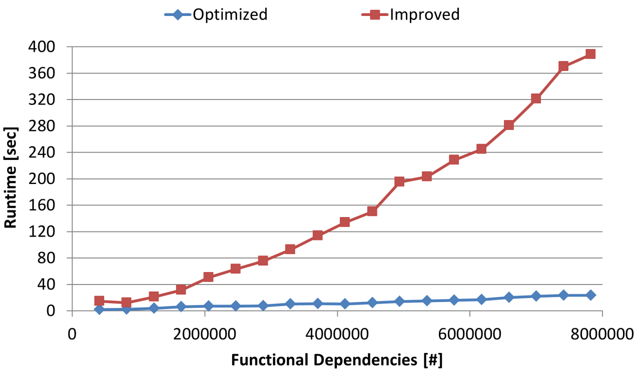


Figure 2: Scaling the number of input FDs for closure calculation.

[4] http://tpc.org/tpch

[5] https://musicbrainz.org

[6] https://hpi.de/naumann/projects/repeatability

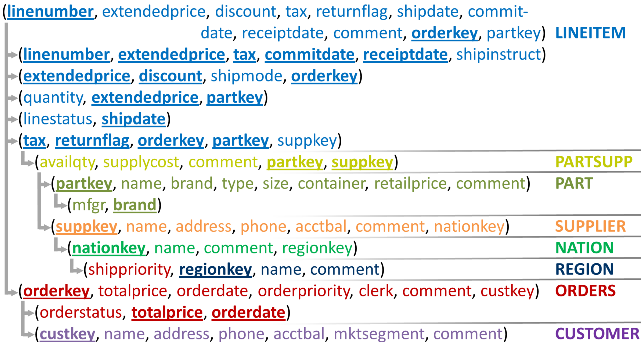

(linenumber, extendedprice, discount, tax, returnflag, shipdate, commitdate, receiptdate, comment, orderkey, partkey)
LINEITEM
(linenumber, extendedprice, tax, commitdate, receiptdate, shipinstruct)
(extendedprice, discount, shipmode, orderkey)
(quantity, extendedprice, partkey)
(linestatus, shipdate)
(tax, returnflag, orderkey, partkey, suppkey)
(availqty, supplycost, comment, partkey, suppkey)
PARTSUPP
(partkey, name, brand, type, size, container, retailprice, comment)
PART
(mfgr, brand)
(suppkey, name, address, phone, acctbal, comment, nationkey)
SUPPLIER
(nationkey, name, comment, regionkey)
NATION
(shippriority, regionkey, name, comment)
REGION
(orderkey, totalprice, orderdate, orderpriority, clerk, comment, custkey)
ORDERS
(orderstatus, totalprice, orderdate)
(custkey, name, address, phone, acctbal, mktsegment, comment)
CUSTOMER

Figure 3: Relations after normalizing TPC-H.

Both runtimes in Figure 2 appear to scale almost linearly with the number of FDs, because the extension costs for each single FD are low due to the efficient index lookups. Nevertheless, the index lookups become more expensive with an increasing number of FDs in the indexes (and they would also become more numerous, if we would increase the number of attributes as well). Because the improved algorithm performs the index lookups more often than the optimized version (i.e. changed loop) and with larger search keys (i.e. LHS and RHS), the optimized version is faster and scales better with the number of FDs: It is from 4 to 16 times faster in this experiment.

8.3 Effectiveness analysis

For a fair effectiveness analysis, we perform the normalization automatically, i.e., without human interaction. Under human supervision, better (but possibly also worse) schemata than presented below can be produced. For the following experiments, we focus on TPC-H and MusicBrainz, because we denormalized these datasets before so that we can use their original schemata as gold standards for their normalization results.

Figure 3 shows the BCNF normalized TPC-H dataset. The color coding indicates the original relations of the different attributes. So we first notice that NORMALIZE almost perfectly restored the original schema: We can identify all original relations in the normalized result. The automatically selected constraints, i.e., keys and foreign keys are all correct w.r.t. the original schema, which is possible because the original schema was snow-flake shaped.

Nevertheless, we also observe two interesting flaws in the automatically normalized schema: First, NORMALIZE decomposed the LINEITEM relation a bit too far; syntactically, the result is correct and perfectly BCNF-conform, but semantically, the splits with only one dependent and more than three foreign key attributes are not reasonable. Second, the attribute shippriority originally belongs to the ORDERS relation but was placed into the REGION relation. This is syntactically a good decision, because the region also determines the shipping priority and putting the attribute into this relation removes more redundant values than putting it into the ORDERS relation.

Figure 4 shows the BCNF-normalized MusicBrainz dataset. Although MusicBrainz has originally no snow-flake schema, NORMALIZE was still able to reconstruct almost all original relations. Only ARTIST_CREDIT_NAME was not reconstructed and its attributes now lie in the semantically related ARTIST relation. Because MusicBrainz is originally not snow-flake shaped, the normalization produced a new top-level relation that represents all many-to-many relationships between artists, places, release labels, and tracks. This top-level relation can be likened to a fact table.

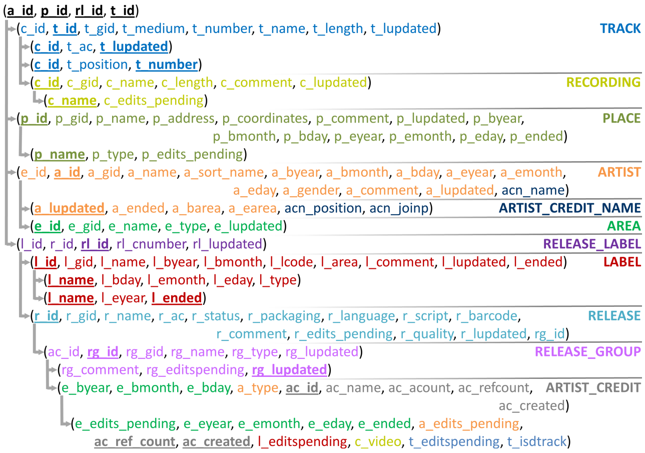

(a_id, p_id, rl_id, t_id)
(c_id, t_id, t_gid, t_medium, t_number, t_name, t_length, t_lupdated)
TRACK
(c_id, t_ac, t_lupdated)
(c_id, t_position, t_number)
(c_id, c_gid, c_name, c_length, c_comment, c_lupdated)
RECORDING
(c_name, c_edits_pending)
(p_id, p_gid, p_name, p_address, p_coordinates, p_comment, p_lupdated, p_byear, p_bmonth, p_bday, p_eyear, p_emonth, p_eday, p_ended)
PLACE
(p_name, p_type, p_edits_pending)
(e_id, a_id, a_gid, a_name, a_sort_name, a_byear, a_bmonth, a_bday, a_eyear, a_emonth, a_eday, a_gender, a_comment, a_lupdated, acn_name)
ARTIST
(a_lupdated, a_ended, a_barea, a_earea, acn_position, acn_joinp)
ARTIST_CREDIT_NAME
(e_id, e_gid, e_name, e_type, e_lupdated)
AREA
(l_id, r_id, rl_id, rl_cnumber, rl_lupdated)
RELEASE_LABEL
(l_id, l_gid, l_name, l_byear, l_bmonth, l_lcode, l_area, l_comment, l_lupdated, l_ended)
LABEL
(l_name, l_bday, l_emonth, l_eday, l_type)
(l_name, l_eyear, l_ended)
(r_id, r_gid, r_name, r_ac, r_status, r_packaging, r_language, r_script, r_barcode, r_comment, r_edits_pending, r_quality, r_lupdated, rg_id)
RELEASE
(ac_id, rg_id, rg_gid, rg_name, rg_type, rg_lupdated)
RELEASE_GROUP
(rg_comment, rg_editspending, rg_lupdated)
(e_byear, e_bmonth, e_bday, a_type, ac_id, ac_name, ac_acount, ac_refcount, ac_created)
ARTIST_CREDIT
(e_edits_pending, e_eyear, e_emonth, e_eday, e_ended, a_edits_pending, ac_ref_count, ac_created, l_editspending, c_video, t_editspending, t_isdtrack)

Figure 4: Relations after normalizing MusicBrainz.

Most mistakes are made for the ARTIST_CREDIT relation, which was the first proposed split. This split took away some attributes from other relations, because these attributes do not contain many values and assigning them to the ARTIST_CREDIT relation makes syntactically sense. A human expert, if involved, would have likely avoided that, because NORMALIZE does report to the user that these attributes are also dependent on other violating FDs LHS attributes. Overall, however, the normalization result is quite satisfactory, keeping in mind that no human was involved in creating it.

We also tested NORMALIZE on various other datasets with similar findings: If datasets have been de-normalized before, we can find the original tables in the proposed schema; if sparsely populated columns exist, these are often moved into smaller relations; and if no human is in the loop, some decompositions become detailed. All results were BCNF-conform and semantically understandable.

9. CONCLUSION

We proposed NORMALIZE, an instance-driven, (semi-) automatic algorithm for schema normalization. The algorithm has shown that functional dependency profiling results of any size can efficiently be used for the specific task of schema normalization. We also presented techniques for guiding the BCNF decomposition algorithm in order to produce semantically good normalization results that also conform to changes of the data.

Our implementation is publicly available at http://hpi.de/naumann/projects/repeatability. It is currently console-based, offering only basic user interaction. Future work shall concentrate on emphasizing the user-in-the-loop, for instance, by employing graphical previews of normalized relations and their connections. We also suggest research on other features for the key and foreign key selection that may yield even better results. Another open research question is how normalization processes should handle dynamic data and errors in the data.

10. REFERENCES

[1] Z. Abedjan, P. Schulze, and F. Naumann. DFD: Efficient functional dependency discovery. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 949–958, 2014.

[2] P. Andritsos, R. J. Miller, and P. Tsaparas. Information-theoretic tools for mining database structure from large data sets. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 731–742, 2004.

[3] C. Beeri and P. A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems (TODS)*, 4(1):30–59, 1979.

[4] P. A. Bernstein. Synthesizing third normal form relations from functional dependencies. *ACM Transactions on Database Systems (TODS)*, 1(4):277–298, 1976.

[5] S. Ceri and G. Gottlob. Normalization of relations and prolog. *Communications of the ACM*, 29(6):524–544, 1986.

[6] E. F. Codd. Derivability, redundancy and consistency of relations stored in large data banks. *IBM Research Report, San Jose, California*, RJ599, 1969.

[7] E. F. Codd. Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909, 1971.

[8] C. J. Date. *Database Design & Relational Theory.* O'Reilly Media, 2012.

[9] J. Diederich and J. Milton. New methods and fast algorithms for database normalization. *ACM Transactions on Database Systems (TODS)*, 13(3):339–365, 1988.

[10] R. Fagin. Normal forms and relational database operators. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 153–160, 1979.

[11] P. A. Flach and I. Savnik. Database dependency discovery: a machine learning approach. *AI Communications*, 12(3):139–160, 1999.

[12] 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.

[13] A. Heise, J.-A. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable discovery of unique column combinations. *Proceedings of the VLDB Endowment*, 7(4):301–312, 2013.

[14] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[15] H. Köhler, S. Link, and X. Zhou. Possible and certain SQL key. *Proceedings of the VLDB Endowment*, 8(11):1118–1129, 2015.

[16] H. Mannila and K.-J. Räihä. Dependency inference. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 155–158, 1987.

[17] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann. Data profiling with Metanome. *Proceedings of the VLDB Endowment*, 8(12):1860–1871, 2015.

[18] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proceedings of the VLDB Endowment*, 8(10):1082–1093, 2015.

[19] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2016.

[20] A. Rostin, O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser. A machine learning approach to foreign key discovery. In *Proceedings of the ACM Workshop on the Web and Databases (WebDB)*, 2009.

Towards Interactive Debugging of Rule-based Entity Matching

Fatemah Panahi Wentao Wu AnHai Doan Jeffrey F. Naughton
Department of Computer Sciences, University of Wisconsin-Madison
{fatemeh, wentaowu, anhai, naughton}@cs.wisc.edu

ABSTRACT

Entity Matching (EM) identifies pairs of records referring to the same real-world entity. In practice, this is often accomplished by employing analysts to iteratively design and maintain sets of matching rules. An important task for such analysts is a "debugging" cycle in which they make a modification to the matching rules, apply the modified rules to a labeled subset of the data, inspect the result, and then perhaps make another change. Our goal is to make this process interactive by minimizing the time required to apply the modified rules. We focus on a common setting in which the matching function is a set of rules where each rule is in conjunctive normal form (CNF). We propose the use of "early exit" and "dynamic memoing" to avoid unnecessary and redundant computations. These techniques create a new optimization problem, and accordingly we develop a cost model and study the optimal ordering of rules and predicates in this context. We also provide techniques to reuse previous results and limit the computation required to apply incremental changes. Through experiments on six real-world data sets we demonstrate that our approach can yield a significant reduction in matching time and provide interactive response times.

1. INTRODUCTION

Entity matching (EM) identifies pairs of records that refer to the same real-world entity. For example, the records (Matthew Richardson, 206-453-1978) and (Matt W. Richardson, 453 1978) may refer to the same person, and (Apple, Cupertino CA) and (Apple Corp, California) refer to the same company. Entity matching is crucial for data integration and data cleaning.

Rule-based entity matching is widely used in practice [2, 13]. This involves analysts designing and maintaining sets of rules. Analysts typically follow an iterative debugging process, as depicted in Figure 1. For example, imagine an e-commerce marketplace that sells products from different vendors. When a vendor submits products from a new category, the analyst writes a set of rules designed to match these products with existing products. He or she then applies these rules to a labeled subset of the data and waits for the results. If the analyst finds errors in the matching output, he or she will refine the rules and re-run them, repeating the above process until the result is of sufficiently high quality.

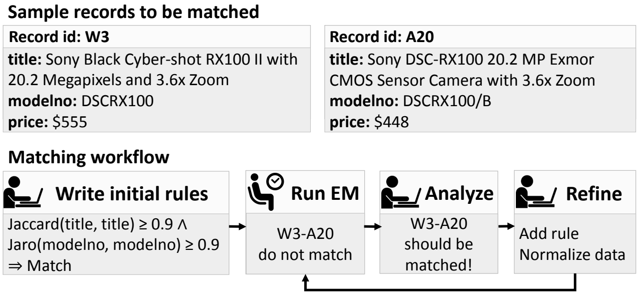

Sample records to be matched
Record id: W3
title: Sony Black Cyber-shot RX100 II with 20.2 Megapixels and 3.6x Zoom
modelno: DSCRX100
price: $555
Record id: A20
title: Sony DSC-RX100 20.2 MP Exmor CMOS Sensor Camera with 3.6x Zoom
modelno: DSCRX100/B
price: $448
Matching workflow
Write initial rules
Jaccard(title, title) ≥ 0.9 ∧ Jaro(modelno, modelno) ≥ 0.9 ⇒ Match
Run EM
W3-A20 do not match
Analyze
W3-A20 should be matched!
Refine
Add rule
Normalize data

Figure 1: A typical matching workflow for analysts.

Our goal is to make this process interactive. This naturally introduces two challenges:

- *Efficiency*. The time that an analyst is idle in the "Run EM" step has to be short. Research shows that when interacting with software, if the response time is greater than one second, the analyst's flow of thought will be interrupted, and if it is greater than 10 seconds, the user will turn their attention to other tasks [12]. Therefore, it is imperative to reduce the idle "waiting" time as much as possible.
- *Maintainability*. The refinement made by the analyst is conceivably incremental. It is therefore desirable for an interactive rule-based entity matching system to maintain matching states between consecutive runs of "Run EM." Although a "stateless" system is easier to implement, e.g., it could just rerun the whole matching algorithm again from the scratch upon each refinement of the rules, it is clearly suboptimal in terms of both runtime efficiency and resource utilization.

In this paper, we take a first step towards interactive debugging of rule-based entity matching. Typically, rule-based entity matching is accomplished by evaluating a boolean matching function for each candidate record pair (for example, $B1$ in Figure 2). In this paper, we follow the approach we have encountered in practice in which this matching function is in Disjunctive Normal Form (DNF). Each disjunction is a rule, and each rule is a conjunction of a set of predicates that evaluate the similarity of two records on one or more attributes, using a similarity function (such as Jaccard or TF-IDF). For example, Jaccard$(a.name, b.name) > 0.7$ is a predicate, where Jaccard$(a.name, b.name)$ is a feature. A record pair is a match if it matches at least one rule.

As was pointed out by Benjelloun et al. [2], and confirmed in our experiments, computing similarity function values dominates the matching time. In view of this, our basic idea is to minimize

Series ISSN: 2367-2005

10.5441/002/edbt.2017.32

the number of similarity function computations as the analyst defines new rules and/or refines existing rules. Specifically, we save all computed similarity function values in memory to avoid redundant computation. We then exploit natural properties of DNF/CNF rule sets that enable "early exit" evaluation to eliminate the need for evaluating many rules and/or predicates for a given candidate pair of records. Moreover, we use "dynamic memoing": we only compute (and save the result of) a feature (i.e., a similarity score) if that predicate result is required by the matching function. (Because of early exit, not all features need to be computed.) This "lazy feature computation" strategy can thus save significant computation cost when there are many possible features but only a few of them are really required by the rule set/data set under consideration.

Although techniques such as "early exit" and "dynamic memoing" are straightforward and ubiquitous in computer science, their application in the context of rule-based entity matching raises an interesting, challenging issue: different evaluation orders of the predicates and rules may lead to significant differences in computational cost. It is then natural to ask the question of optimal ordering of predicates and rules. We further study this problem in detail. We show that the optimization problem under our setting is NP-hard, and we propose two greedy solutions based on heuristic optimization criteria. In our experiments with six real-world datasets, we show that the greedy solutions can indeed produce orderings that significantly reduce runtime compared to random ones.

So far, we have been focusing on the "efficiency" aspect of interactive entity matching. Since the elements (e.g., features, predicates, or rules) involved in matching change frequently as the analyst iteratively refines the rule set, the "maintainability" aspect is of equal importance. We therefore further develop incremental matching techniques to avoid rerunning matching from scratch after each change. Specifically, we discuss four fundamental cases: add/remove a predicate and add/remove a rule. We show how easy it is to integrate incremental matching into our framework. We also show via experiments that our incremental solutions can reduce matching time by orders of magnitude.

1.1 Related Work

Our work differs from previous work in several ways. Previous work on efficiently running rule-based entity resolution [2] assumes that each predicate is a black box, and thus memoing of similarity function results is not possible. In our experience in an industrial setting, these predicates are often not black boxes — rather, they are explicitly presented in terms of similarity functions, attributes, and thresholds. On the other hand, the traditional definition of the EM workflow, as described in [3, 5], assumes that *all* similarity values for *all* pairs are *precomputed* before the matching step begins. This makes sense in a batch setting in which a static matching function has been adopted, and the task is to apply this function to a set of candidate record pairs. However, in this paper we are concerned with the exploratory stage of rule generation, where at the outset the matching function is substantially unknown. In such settings the combinatorial explosion of potential attributes pairs, potential similarity functions, and candidate pairs can render such full precomputation infeasible.

Even in small problem instances in which full precomputation may be feasible, it can impose a substantial lag time between the presentation of a new matching task and the time when the analyst can begin working. This lag time may not be acceptable in practical settings where tens of matching tasks may be created every day [7] and the analyst wants to start working on high priority tasks immediately. Finally, during the matching process, an analyst may perform cleaning operations, normalization, and attribute extractions on the two input tables. The analyst might also introduce new similarity functions. In any of these situations, it is not possible to precompute all features a priori.

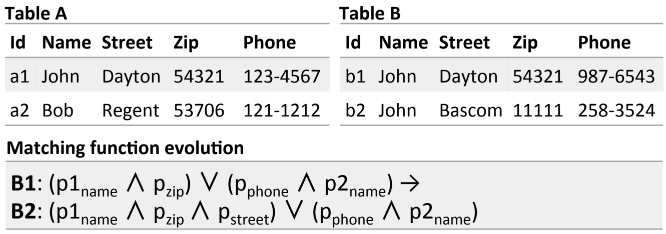


Table A

Id	Name	Street	Zip	Phone
a1	John	Dayton	54321	123-4567
a2	Bob	Regent	53706	121-1212

Table B

Id	Name	Street	Zip	Phone
b1	John	Dayton	54321	987-6543
b2	John	Bascom	11111	258-3524

Matching function evolution

B1: $(p1_{name} \wedge p_{zip}) \vee (p_{phone} \wedge p2_{name}) \rightarrow$

B2: $(p1_{name} \wedge p_{zip} \wedge p_{street}) \vee (p_{phone} \wedge p2_{name})$

Figure 2: Tables A, B to be matched and example matching functions. Function B1 evolves to B2.

Previous work on incrementally evaluating the matching function when the logic evolves assumes that we evaluate all predicates for all pairs and materialize the matching result for each predicate [15]. Because we use early exit, our information about the matching results for each predicate is not complete. As a result, this solution is not directly applicable in our setting.

In other related work, Dedoop (abbreviation for "Deduplication with Hadoop") [9] seeks to improve performance for general, large, batch entity matching tasks through the exploitation of parallelism available in Hadoop. By contrast, our work focuses on interactive response for rule-based entity matching where the matching function is composed of many rules that evolve over time. Exploring the application of parallelism as explored in Dedoop to our context is an interesting area for future work.

Our work is also related to [14]. In that work, the user provides a set of rule templates and a set of labeled matches and non-matches, the system then efficiently searches a large space of rules (that instantiate the rule templates) to find rules that perform best on the labeled set (according to an objective function). That work also exploits the similarities among the rules in the space. But it does so to search for the best set of rules efficiently. In contrast, we exploit rule similarities to support interactive debugging.

Finally, our work is related to the Magellan project, also at UW-Madison [10]. That project proposes to perform entity matching in two stages. In the development stage, the user iteratively experiments with data samples to find an accurate EM workflow. Then in the production stage the user executes that workflow on the entirety of data. If the user has decided to use a rule-based approach to EM, then in the development stage he or she will often have to debug the rules, which is the focus of this paper. This work thus fits squarely into the development stage of the Magellan approach.

In the following we start with a motivating example, describe our approach to try to achieve interactive response times, and present experimental results of our techniques on real world data sets.

2. MOTIVATING EXAMPLE

To motivate and give an overview of our approach, consider the following example. Our task is to match Table A and Table B shown in Figure 2 to find records that refer to the same person. We have four candidate pairs of records: $\{a1b1, a1b2, a2b1, a2b2\}$. Assume our matching function is $B1$. Intuitively, $B1$ says that if the name and zipcode of two records are similar, or if the phone number and name of two records are similar, then they match. Here $p1_{name}$ and $p2_{name}$, for example, compute the similarity score $Jaccard(a.name, b.name)$ and then compare this value to different thresholds, respectively, as we will see below.[1] For this ex-

[1] In practice we often compute Jaccard over the sets of q-grams of the two names, e.g., where $q = 3$; here for ease of exposition we

ample, $B1$ will return true for $a1b1$ and false for the rest of the candidate pairs.

A simple way to accomplish matching is to evaluate every predicate for every candidate pair. To evaluate a predicate, we compute the value of the similarity function associated with that predicate and compare it to a threshold. For the candidate pair $a2b1$, we would compute 4 similarity values.

This is unnecessary because once a predicate in a rule evaluates to false, we can skip the remaining predicates. Similarly, once a rule evaluates to true, we can skip the rest of the rules and therefore finish matching for that pair. We call this strategy "early exit," which saves unnecessary predicate evaluations. For instance, consider the candidate pair $a2b1$ again. Suppose that the predicate $p1_{name}$ is

$$\text{Jaccard}(a.name, b.name) \geq 0.9.$$

Since the Jaccard similarity of the two names is 0, $p1_{name}$ will return false for this candidate pair. Further assume that p_{phone} performs an equality check and thus returns 0 as well. We then do not need to evaluate p_{zip} and $p2_{name}$ to make a decision for this pair. Therefore, for this candidate pair, "early exit" reduces the number of similarity computations from 4 to 2.

Since the same similarity function may be applied to a candidate pair in multiple rules and predicates, we "memo" each similarity value once it has been computed. If a similarity function appears in multiple predicates, only the first evaluation of the predicate incurs a computation cost, while subsequent evaluations only incur (much cheaper) lookup costs. We call this strategy "dynamic memoing." Continuing with our example, suppose $p2_{name}$ is

$$\text{Jaccard}(a.name, b.name) \geq 0.7.$$

Then for $a1b2$ this predicate only involves a lookup cost.

When using *early exit* and *dynamic memoing*, different orders of the predicates/rules will make a difference in the overall matching cost. Once again consider the candidate pair $a2b1$. If we change the order of predicates in $B1$ to

$$(p1_{name} \wedge p_{zip}) \vee (p2_{name} \wedge p_{phone}),$$

the output of the matching function will not change. However, it reduces the matching cost to one computation for $p1_{name}$ plus one lookup for $p2_{name}$. This raises a novel optimization problem that we study in Section 5.

Finally, we take into account the fact that, as the matching function's logic evolves, the changes to the function are often *incremental*. We can then store results of a previous EM run, and as the EM logic evolves, use those to save redundant work for the next EM iterations. As an example, imagine the case where the matching function $B1$ evolves to $B2$. Since $B2$ is *stricter* than $B1$, we only need to evaluate p_{street} for the pairs that were matched by $B1$ to verify if they still match. For our example, this means that we only need to evaluate $B1$ for $a1b1$ among the four pairs.

3. PRELIMINARIES

The input to the entity matching (EM) workflow is two tables A, B with a set of records $\{a_1 \dots a_n\}$, $\{b_1 \dots b_m\}$ respectively. The goal of EM is to find all record pairs a_ib_j that refer to the same entity. Given table A with m records and table B with n records, there are $m \times n$ potential matches. Even with moderate-size tables, the total number of potential matches could be very large. Many potential matches obviously do not match and can be eliminated from consideration easily. That is the purpose of a *blocking* step, which typically precedes a more detailed matching phase.

For example, suppose that each product has a category attribute (e.g., clothing or electronics). We can assume that products from different categories are non-matches. This reduces the task to finding matching products within the same category. We refer to the set of potential matches left after the blocking step as the *candidate record pairs* or *candidate pairs* in the rest of this paper.

Each candidate record pair is evaluated by a Boolean *matching function* B, which takes in two records and returns true or false. We assume that B is commutative, i.e.,

$$\forall a_ib_j, B(a_i, b_j) = B(b_j, a_i).$$

We assume that each matching function is in disjunctive normal form (DNF). We refer to each disjunct as a *rule*. For example, our matching function $B1$ (Figure 2) is composed of two rules.

Such a matching function is composed of only "positive" rules, as they say what matches, not what does not match. In our experience, this is a common form of matching function used in the industry. Reasons for using only positive rules include ease of rule generation, comprehensibility, ease of debugging, and commutativity of rule application.

Each rule is a conjunction of a set of *predicates*. Each predicate compares the value of a *feature* for a candidate pair with a threshold. A feature in our context is a similarity function computed over attributes from the two tables. Similarity functions can be as simple as exact equality, or as complex as arbitrary user-defined functions requiring complex pre-processing and logic.

The matching result is composed of the return value of the matching function for each of the candidate pairs. In order to evaluate the quality of matching, typically a sample of the candidate pairs is chosen and manually labeled as match or non-match based on domain knowledge. The matching results for the sample is then compared with the correct labels to get an estimate of the quality of matching (e.g., *precision* and *recall*).

4. EARLY EXIT + DYNAMIC MEMOING

In this section, we first briefly present the details of *early exit* and *dynamic memoing*. Although the ideas are pretty straightforward, we choose to describe them in an algorithmic way for clarity. The notation used in describing these algorithms will also be used throughout the rest of the paper when we discuss optimal predicate/rule ordering and incremental algorithms. To analyze the costs of various algorithms covered in this section, we further develop a cost model. It is also the basis for the next section when we study the optimal predicate/rule ordering problem.

4.1 Baselines

We study two baseline approaches in this subsection. In the following, for a given rule r, we use $\text{predicate}(r)$ and $\text{feature}(r)$ to denote the set of predicates and features r includes.

4.1.1 The Rudimentary Baseline

The first baseline algorithm simply evaluates every predicate in the matching function for every candidate pair. Each predicate is considered as a black box and any similarity value used in the predicate is computed from scratch. The results of the predicates (true or false) are then passed on to the rules, and the outputs of the rules passed on to the matching function to determine the matching status. Algorithm 1 presents the details of this baseline.

will assume that Jaccard scores are computed over the set of words of the two names.

Algorithm 1: The rudimentary baseline.

```
Input: $B$, the matching function; $\mathcal{C}$, candidate pairs
Output: $\{(c, x)\}$, where $c \in \mathcal{C}$ and $x \in \{M, U\}$ ($M$ means a
        match and $U$ means an unmatch)
Let $\mathcal{R}$ be the CNF rules in $B$;
Mark all $c \in \mathcal{C}$ with $U$;
foreach $c \in \mathcal{C}$ do
    foreach $r \in \mathcal{R}$ do
        foreach $p \in \text{predicate}(r)$ do
            Evaluate $p$;
        end
        Evaluate $r = \bigwedge_{p \in \text{predicate}(r)} p$;
    end
    Mark $c$ with $M$ if $B = \bigvee_{r \in \mathcal{R}} r$ is true;
end
```

4.1.2 The Precomputation Baseline

This algorithm precomputes all feature values involved in the predicates before performing matching. Algorithm 2 presents the details. As noted in the introduction, full precomputation may not be feasible or desirable in practice, but we present it here as a point of comparison. We store precomputed values as a hash table mapping pairs of attribute values to similarity function outputs.

Algorithm 2: The precomputation baseline.

```
Input: $B$, the matching function; $\mathcal{C}$, candidate pairs
Output: $\{(c, x)\}$, where $c \in \mathcal{C}$ and $x \in \{M, U\}$ ($M$ means a
        match and $U$ means an unmatch)
Let $\mathcal{R}$ be the CNF rules in $B$;
Let $\mathcal{F} = \bigcup_{r \in \mathcal{R}} \text{feature}(r)$;
Let $\Gamma = \{(c, f, v)\}$ be a $|\mathcal{C}| \times |\mathcal{F}|$ array that stores the value $v$
  of each $f \in \mathcal{F}$ for each $c \in \mathcal{C}$;
foreach $c \in \mathcal{C}$ do
    foreach $f \in \mathcal{F}$ do
        Compute $v$ and store $(c, f, v)$ in $\Gamma$;
    end
end
Run Algorithm 1 by looking up feature values from $\Gamma$ when
  evaluating predicates;
```

4.2 Early Exit

Both baselines discussed above ignore the properties of the matching function B. Given that B is in DNF, if one of the rules returns true, B will return true. Similarly, because each rule in B is in CNF, a rule will return false if one of its predicate returns false. Therefore, we do not need to evaluate all the predicates and rules. Algorithm 3 uses this idea. The "breaks" in lines 8 and 12 are the "early exits" in this algorithm.

4.3 Dynamic Memoing

We can combine the precomputation of the second baseline with early exit. That is, instead of precomputing everything up front, we postpone the computation of a feature until it is encountered during matching. Once we have computed the value of a feature, we store it so following references of this feature only incur lookup costs. We call this strategy "dynamic memoing," or "lazy feature computation." Algorithm 4 presents the details.

4.4 Cost Modeling and Analysis

In this subsection, we develop simple cost models to use in rule and predicate ordering decisions studied in Section 5. In the following discussion, we use $\text{cost}(p)$ to denote the cost of evaluating a predicate p. Let $\mathcal{C}$ be the set of all candidate pairs. Moreover, let F be the set of all features involved in the matching function, and we use $\text{cost}(f)$ to denote the computation cost of a feature f. Furthermore, we use δ to represent the lookup cost.

Algorithm 3: Early exit.

```
   Input: $B$, the matching function; $\mathcal{C}$, candidate pairs
   Output: $\{(c, x)\}$, where $c \in \mathcal{C}$ and $x \in \{M, U\}$ ($M$ means a
           match and $U$ means an unmatch)
 1 Let $\mathcal{R}$ be the CNF rules in $B$;
 2 Mark all $c \in \mathcal{C}$ with $U$;
 3 foreach $c \in \mathcal{C}$ do
 4     foreach $r \in \mathcal{R}$ do
 5         $r$ is true;
 6         foreach $p \in \text{predicate}(r)$ do
 7             if $p$ is false then
 8                 $r$ is false; break;
 9             end
10         end
11         if $r$ is true then
12             Mark $c$ with $M$; break;
13         end
14     end
15 end
```

4.4.1 The Rudimentary Baseline

The cost of Algorithm 1 can be represented as:

$$C_1 = \sum\nolimits_{c \in \mathcal{C}} \sum\nolimits_{r \in \mathcal{R}} \sum\nolimits_{p \in \text{predicate}(r)} \text{cost}(p).$$

In our running example in the introduction, the cost of making a decision for the pair $a1b2$ is then

$$\text{cost}(p1_{name}) + \text{cost}(p_{zip}) + \text{cost}(p_{phone}) + \text{cost}(p2_{name}).$$

4.4.2 The Precomputation Baseline

Suppose that each feature f appears $\text{freq}(f)$ times in the matching function. Then the cost of the precomputation baseline (Algorithm 2) is

$$C_2 = \sum\nolimits_{c \in \mathcal{C}} \sum\nolimits_{f \in F} (\text{cost}(f) + \text{freq}(f)\delta).$$

In our running example this means that, for pair $a1b2$ and matching function $B1$, we would need to precompute three similarity values and look up four. Note that this requires knowing $\text{cost}(f)$ — in our implementation, as discussed in our experimental results, we use an estimate of $\text{cost}(f)$ obtained by evaluating f over a sample of the candidate pairs.

4.4.3 Early Exit

To compute the cost of early exit (Algorithm 3), we further introduce the probability $\text{sel}(p)$ that the predicate p will return true for a given candidate pair (i.e., the *selectivity* of p). In our implementation, we use an estimate of $\text{sel}(p)$ obtained by evaluating p over a sample of the candidate pairs.

Given this estimate for $\text{sel}(p)$, suppose that we have a rule r with m predicates p_1, ..., p_m. The expected cost of evaluating r for a (randomly picked) candidate pair is then

$$\begin{aligned} \text{cost}(r) &= \text{cost}(p_1) + \text{sel}(p_1)\,\text{cost}(p_2) + \cdots \\ &\quad + \text{sel}\Big(\bigwedge\nolimits_{j=1}^{m-1} p_j\Big)\,\text{cost}(p_m), \end{aligned} \tag{1}$$

Algorithm 4: Early exit with dynamic memoing.

```
Input: $B$, the matching function; $\mathcal{C}$, candidate pairs
Output: $\{(c, x)\}$, where $c \in \mathcal{C}$ and $x \in \{M, U\}$ ($M$ means a
        match and $U$ means an unmatch)
Let $\mathcal{R}$ be the CNF rules in $B$;
Let $\Gamma$ be the feature values computed; $\Gamma \leftarrow \emptyset$;
Mark all $c \in \mathcal{C}$ with $U$;
foreach $c \in \mathcal{C}$ do
    foreach $r \in \mathcal{R}$ do
        $r$ is true;
        foreach $p \in \mathrm{predicate}(r)$ do
            Let $f$ be the feature in $p$;
            if $f \notin \Gamma$ then
                Compute $f$; $\Gamma \leftarrow \Gamma \cup \{f\}$;
            else
                Read the value of $f$ from $\Gamma$;
            end
            if $p$ is false then
                $r$ is false; break;
            end
        end
        if $r$ is true then
            Mark $c$ with $M$; break;
        end
    end
end
```

because we only need to evaluate p_j if $p_1, ..., p_{j-1}$ are all evaluated to be true. Similarly, we can define the selectivity of the rule r as

$$\mathrm{sel}(r) = \mathrm{sel}(\bigwedge_{j=1}^{m} p_j).$$

Suppose that we have n rules $r_1, ..., r_n$. The expected cost of the early exit strategy (Algorithm 3) is then

$$\begin{aligned} C_3 &= \mathrm{cost}(r_1) + (1 - \mathrm{sel}(r_1))\,\mathrm{cost}(r_2) + \cdots \\ &\quad + (1 - \mathrm{sel}(\bigvee_{i=1}^{n-1} r_i))\,\mathrm{cost}(r_n). \end{aligned}$$

4.4.4 Early Exit with Dynamic Memoing

The expected cost of early exit with dynamic memoing (Algorithm 4) can be estimated in a similar way. The only difference is that we need to further know the probability that a feature is present in the memo. Specifically, suppose that a feature can appear at most once in a rule. Let $\alpha(f, r_i)$ be the probability that a feature f is present in the memo after evaluating r_i. The expected cost of computing f when evaluating r_i is then

$$E[\mathrm{cost}(f)] = (1 - \alpha(f, r_{i-1})\,\mathrm{cost}(f) + \alpha(f, r_{i-1})\delta. \quad (2)$$

The expected cost C_4 of Algorithm 4 is obtained by replacing all $\mathrm{cost}(p)$'s in Equation 1 by their expected costs in Equation 2.

Let $\mathrm{prev}(f, r_i)$ be the predicates in the rule r_i that appear before f. We then have

$$\alpha(f, r_i) = (1 - \alpha(f, r_{i-1}))\,\mathrm{sel}(\bigwedge_{p \in \mathrm{prev}(f, r_i)} p) + \alpha(f, r_{i-1}).$$

Based on our assumption, different predicates in the same rule contain different features. If we further assume that predicates with different features are independent, it then follows that

$$\alpha(f, r_i) = (1 - \alpha(f, r_{i-1})) \prod_{p \in \mathrm{prev}(f, r_i)} \mathrm{sel}(p) + \alpha(f, r_{i-1}).$$

Notation	Description
$\mathrm{cost}(X)$	cost of X (X is a feature/predicate/rule)
δ	the lookup cost
$\mathrm{freq}(f)$	frequency of feature f
$\mathrm{predicate}(r)$	predicates of rule r
$\mathrm{feature}(X)$	features of X (X can be a predicate/rule)
$\mathrm{sel}(X)$	selectivity of X (X can be a predicate/rule)
$\mathrm{prev}(f, r)$	features/predicates in rule r before feature f
$\mathrm{predicate}(f, r)$	predicates in rule r that have feature f
$\mathrm{reduction}(r)$	overall cost reduction by execution of rule r
$\mathrm{cache}(f, r)$	chance that f is in the memo after running r

Table 1: Notation used in cost modeling and optimal rule ordering.

Note that the initial condition satisfies

$$\alpha(f, r_1) = \prod_{p \in \mathrm{prev}(f, r_1)} \mathrm{sel}(p).$$

We therefore have obtained an inductive procedure for estimating $\alpha(f, r_i)$ $(1 \leq i \leq n)$. Clearly, $\alpha(f, r_i) = \alpha(f, r_{i-1})$ if $f \notin \mathrm{feature}(r_{i-1})$. So we can focus on the rules that contain f.

5. OPTIMAL ORDERING

Our goal in this section is to develop techniques to order rule and predicate evaluation to minimize the total cost of matching function evaluation. This may sound familiar, and indeed it is — closely related problems have been studied previously in related settings (see, for example, [1, 8]). However, our problem is different and unfortunately more challenging due to the interaction of early exit evaluation with dynamic memoing.

5.1 Notation

Table 1 summarizes notation used in this section. Some of the notation has been used when discussing the cost models.

5.2 Problem Formulation

We briefly recap an abstract version of the problem. We have a set of rules $\mathcal{R} = \{r_1, ..., r_n\}$. Each rule is in CNF, with each clause containing exactly one predicate. A pair of records is a match if any rule in $\mathcal{R}$ evaluates to true. Therefore, $\mathcal{R}$ is a disjunction of rules:

$$R = r_1 \vee r_2 \vee \cdots \vee r_n.$$

Consider a single rule

$$r = p_1 \wedge p_2 \wedge \cdots \wedge p_m.$$

We are interested in the minimum expected cost of evaluating r with respect to different orders (i.e., permutations) of the predicates $p_1, ..., p_m$.

Given a specific order of the predicates, the expected cost of r can be expressed as

$$\begin{aligned} \mathrm{cost}(r) &= \mathrm{cost}(p_1) + \mathrm{sel}(p_1)\,\mathrm{cost}(p_2) + \cdots \quad (3) \\ &\quad + \mathrm{sel}(\bigwedge_{j=1}^{m-1} p_j)\,\mathrm{cost}(p_m). \end{aligned}$$

Similarly, given a specific order of the rules, the expected cost of evaluating R, as was in Section 4.4.3, is

$$\begin{aligned} \mathrm{cost}(R) &= \mathrm{cost}(r_1) + (1 - \mathrm{sel}(r_1))\,\mathrm{cost}(r_2) + \cdots \quad (4) \\ &\quad + (1 - \mathrm{sel}(\bigvee_{i=1}^{n-1} r_i))\,\mathrm{cost}(r_n). \end{aligned}$$

We want to minimize $\mathrm{cost}(R)$.

5.3 Independent Predicates and Rules

The optimal ordering problem is not difficult when independence of predicates/rules holds. We start by considering the optimal ordering of the predicates in a single rule r. If the predicates are independent, Equation 3 reduces to

$$\begin{aligned}\mathrm{cost}(r) &= \mathrm{cost}(p_1) + \mathrm{sel}(p_1)\,\mathrm{cost}(p_2) + \cdots \\ &\quad + \mathrm{sel}(p_1)\cdots\mathrm{sel}(p_{m-1})\,\mathrm{cost}(p_m).\end{aligned}$$

The following lemma is well known for this case (e.g., see Lemma 1 of [8]):

LEMMA 1. *Assume that the predicates in a rule r are independent.* $\mathrm{cost}(r)$ *is minimized by evaluating the predicates in ascending order of the metric:*

$$\mathrm{rank}(p_i) = (\mathrm{sel}(p_i) - 1)/\,\mathrm{cost}(p_i) \qquad (\textit{for } 1 \le i \le m).$$

We next consider the optimal ordering of the rules by assuming that the rules are independent. We have the following similar result.

THEOREM 1. *Assume that the predicates in all the rules are independent.* $\mathrm{cost}(R)$ *is minimized by evaluating the rules in ascending order of the metric:*

$$\mathrm{rank}(r_j) = -\frac{\mathrm{sel}(r_j)}{\mathrm{cost}(r_j)} = -\frac{\prod_{p\in\mathrm{predicate}(r_j)}\mathrm{sel}(p)}{\mathrm{cost}(r_j)}.$$

Here $\mathrm{cost}(r_j)$ *is computed by using Equation 3 with respect to the order of predicates specified in Lemma 1.*

PROOF. By De Morgan's laws, we have

$$R = r_1 \vee \cdots \vee r_n = \neg(\bar{r}_1 \wedge \cdots \wedge \bar{r}_n).$$

Define $r'_j = \bar{r}_j$ for $1 \le j \le n$ and $R' = \neg R$. It follows that

$$R' = r'_1 \wedge \cdots \wedge r'_n.$$

This means, to evaluate R, we only need to evaluate R', and then take the negation. Since R' is in CNF, based on Lemma 1, the optimal order is based on

$$\mathrm{rank}(r'_j) = (\mathrm{sel}(r'_j) - 1)/\,\mathrm{cost}(r'_j) \qquad (\text{for } 1 \le j \le n).$$

We next compute $\mathrm{sel}(r'_j)$ and $\mathrm{cost}(r'_j)$. First, we have

$$\mathrm{sel}(r'_j) = 1 - \mathrm{sel}(r_j) = 1 - \prod\nolimits_{p\in\mathrm{predicate}(r_j)}\mathrm{sel}(p),$$

by the independence of the predicates. Moreover, we simply have $\mathrm{cost}(r'_j) = \mathrm{cost}(r_j)$, because we can evaluate r'_j by first evaluating r_j and then taking the negation. Therefore, it follows that

$$\mathrm{rank}(r_j) = \mathrm{rank}(r'_j) = -\frac{\mathrm{sel}(r_j)}{\mathrm{cost}(r_j)} = -\frac{\prod_{p\in\mathrm{predicate}(r_j)}\mathrm{sel}(p)}{\mathrm{cost}(r_j)}.$$

This completes the proof of the theorem. □

Recall that in our implementation we compute feature costs and selectivity by sampling a set of record pairs and compute the costs and selectivities on the sample. So far, we have implicitly assumed that memoing is not used.

5.4 Correlated Predicates and Rules

We now consider the question when memoing is used. This introduces dependencies so Lemma 1 and Theorem 1 no longer hold.

Let us start with one single rule r. We introduce a *canonical form* of r by "grouping" together predicates that share common features. Formally, for a predicate p, let $\mathrm{feature}(p)$ be the feature it refers to. Furthermore, define

$$\mathrm{feature}(r) = \cup_{p\in\mathrm{predicate}(r)}\{\mathrm{feature}(p)\}.$$

Given a rule r and a feature $f \in \mathrm{feature}(r)$, let

$$\mathrm{predicate}(f, r) = \bigwedge\nolimits_{p\in\mathrm{predicate}(r)\wedge\mathrm{feature}(p)=f} p.$$

We can then write the rule r as

$$r = \bigwedge\nolimits_{f\in\mathrm{feature}(r)} \mathrm{predicate}(f, r). \tag{5}$$

Since we only consider predicates of the form $A \ge a$ or $A \le a$ where A is a feature and a is a constant threshold, it is reasonable to assume that each rule does not contain redundant predicates/features. As a result, each group $\mathrm{predicate}(f, r)$ can contain at most one predicate of the form $A \ge a$ and/or $A \le a$. Based on this observation, we have the following simple result.

LEMMA 2. $\mathrm{cost}(\mathrm{predicate}(f, r))$ *is minimized by evaluating the predicates in ascending order of their selectivities.*

PROOF. Remember that $\mathrm{predicate}(f, r)$ contains at most two predicates p_1 and p_2. Note that, the costs of the predicates follow the pattern c, c' if memoing is used, regardless of the order of the predicates in $\mathrm{predicate}(f, r)$. Here c and c' are the costs of directly computing the feature or looking it up from the memo ($c > c'$). As a result, we need to decide which predicate to evaluate first. This should be the predicate with the lower selectivity. To see this, without loss of generality let us assume $\mathrm{sel}(p_1) < \mathrm{sel}(p_2)$. The overall cost of evaluating p_1 before p_2 is then

$$C_1 = c + \mathrm{sel}(p_1)c',$$

whereas the cost of evaluating p_2 before p_1 is

$$C_2 = c + \mathrm{sel}(p_2)c'.$$

Clearly, $C_1 < C_2$. This completes the proof of the lemma. □

Since the predicates in different groups are independent, by applying Lemma 1 we get the following result.

LEMMA 3. $\mathrm{cost}(r)$ *is minimized by evaluating the predicate groups in ascending order of the following metric:*

$$\mathrm{rank}(\mathrm{predicate}(f, r)) = \frac{\mathrm{sel}(\mathrm{predicate}(f, r)) - 1}{\mathrm{cost}(\mathrm{predicate}(f, r))}.$$

Here $\mathrm{cost}(\mathrm{predicate}(f, r))$ *is computed by using Equation 3 with respect to the order of predicates specified in Lemma 2.*

Now let us move on to the case in which there are multiple rules whose predicates are not independent. Unfortunately, this optimization problem is in general NP-hard. We can prove this by reduction from the classic traveling salesman problem (TSP) as follows. Let the rules be vertices of a complete graph G. For each pair of rules r_i and r_j, define the cost $c(i, j)$ of the edge (r_i, r_j) to be the execution cost of r_j if it immediately follows r_i. Note that here we have simplified our problem by assuming that the cost of r_j only depends on its predecessor r_i. Under this specific setting, our problem of finding the optimal rule order is equivalent to seeking a Hamiltonian cycle with minimum total cost in G, which is NP-hard. Moreover, it is known that a constant-factor approximation algorithm for TSP is unlikely to exist unless P equals NP (e.g., see Theorem 35.3 of [4]). Therefore, in the following we seek heuristic approaches based on various greedy strategies.

Algorithm 5: A greedy algorithm based on expected costs of rules.

```
   Input: $\mathcal{R} = \{r_1, ..., r_n\}$, a set of CNF rules
   Output: $\mathcal{R}^\pi$, execution order of the rules
1  Let $Q$ be a priority queue $\langle(\mathrm{cost}(r), r)\rangle$ of the rules;
2  $\mathcal{R}^\pi \leftarrow \emptyset$;
3  foreach $r \in \mathcal{R}$ do
4      Order $\mathrm{predicate}(r)$ according to Lemma 3;
5      Compute $\mathrm{cost}(r)$ based on this order;
6      Insert $(\mathrm{cost}(r), r)$ into $Q$;
7  end
8  while $Q$ is not empty do
9      $r_{\min} \leftarrow \mathrm{ExtractMin}(Q)$;
10     Add $r_{\min}$ into $\mathcal{R}^\pi$;
11     foreach $(\mathrm{cost}(r), r) \in Q$ do
12         Update $\mathrm{cost}(r)$ by assuming that $r$ immediately
           follows $r_{\min}$;
13     end
14 end
15 return $\mathcal{R}^\pi$;
```

5.4.1 Greedy Algorithms

We now need to further order the rules by considering the overhead that can be saved by memoing. By Lemma 3, the predicates in each rule can be locally optimally ordered. Note that each order of the rules induces a global order over the (bag of) predicates. However, the selectivities of the predicates are no longer independent, because predicates in different rules may share the same feature. Furthermore, the costs of predicates are no longer constants due to memoing. In fact, they even depend on the positions of the predicates in their global order. In other words, the costs of predicates depend on the order of the rules (recall the cost model in Section 4.4.4). Hence we are not able to apply Lemma 1 or Theorem 1 in this context.

Nonetheless, intuitively, a predicate should tend to have priority if it is very selective (returns true for very few pairs) and small cost, since it will eliminate many pairs cheaply. On the other hand, a rule should tend to have priority if it is not very selective (returns true for many candidate pairs) and small cost, since it contributes many matches cheaply. Our first algorithm then uses this intuition in a greedy strategy by picking the rule with the minimum expected cost. The details of this algorithm are presented in Algorithm 5. Note that when we update $\mathrm{cost}(r)$ at line 12, we use the cost model developed in Section 4.4.4, which considers the effect of memoing, by assuming that r will be the immediate successor of $r_{\min}$.

Algorithm 5 only considers the expected costs of the rules if they are the first to be run among the remaining rules. Some rules may have slightly high expected costs but significant long-term impact on overall cost reduction. Algorithm 5 does not consider this and thus may overlook these rules. We therefore further consider a different metric that is based on the rules that can be affected if a rule is executed. This gives our second greedy algorithm.

In the following, we use $\mathrm{reduction}(r)$ to represent the overall cost that can be saved by the execution of the rule r, and use $\mathrm{cache}(f, r)$ to represent the probability that a feature f is in the memo after the execution of r. For two features f_1 and f_2 in r, we write $f_1 < f_2$ if f_1 appears before f_2 in the order of predicate groups specified by Lemma 3. Following Section 4.4.4, we redefine $\mathrm{prev}(f, r)$ to be the *features* that appear before f in r, namely,

$$\mathrm{prev}(f, r) = \{f' \in \mathrm{feature}(r) \wedge f' < f\}.$$

If we write r as it is in Equation 5, then

$$\mathrm{sel}(\mathrm{prev}(f, r)) = \prod\nolimits_{f' \in \mathrm{prev}(f,r)} \mathrm{sel}(\mathrm{predicate}(f', r)) \tag{6}$$

is the selectivity of (conjunction of) the predicates appearing before f in r. Here we have abused notation because $\mathrm{prev}(f, r)$ is a set of features rather than a predicate. Basically, $\mathrm{sel}(\mathrm{prev}(f, r))$ is the chance that the feature f needs to be computed (by either direct computation or cache lookup) when executing r. We further define $\mathrm{prev}(r)$ to be the rule executed right before r. It then follows that

$$\begin{aligned}\mathrm{cache}(f, r) &= (1 - \mathrm{cache}(f, \mathrm{prev}(r)))\,\mathrm{sel}(\mathrm{prev}(f, r)) \\ &\quad + \mathrm{cache}(f, \mathrm{prev}(r)).\end{aligned}$$

Next, define $\mathrm{contribution}(r', r)$ to be the reduced cost of r' by executing the rule r before the rule r'. Define $\mathrm{contribution}(r', r, f)$ to be the reduced cost due to the feature f. Let $\mathrm{feature}(r', r) = \mathrm{feature}(r') \cap \mathrm{feature}(r)$. Clearly,

$$\mathrm{contribution}(r', r) = \sum\nolimits_{f \in \mathrm{feature}(r',r)} \mathrm{contribution}(r', r, f).$$

We now consider how to compute $\mathrm{contribution}(r', r, f)$. If we do not run r before r', the expected cost of evaluating f in r' is then

$$\begin{aligned}\mathrm{cost}_1(f, r') &= \mathrm{sel}(\mathrm{prev}(f, r'))\big[\,\mathrm{cache}(f, \mathrm{prev}(r))\delta \\ &\quad +(1 - \mathrm{cache}(f, \mathrm{prev}(r)))\,\mathrm{cost}(f)\big],\end{aligned}$$

whereas if we run r before r' the cost becomes

$$\begin{aligned}\mathrm{cost}_2(f, r') &= \mathrm{sel}(\mathrm{prev}(f, r'))\big[\,\mathrm{cache}(f, r)\delta \\ &\quad +(1 - \mathrm{cache}(f, r))\,\mathrm{cost}(f)\big].\end{aligned}$$

It then follows that

$$\begin{aligned}\mathrm{contribution}(r', r, f) &= \mathrm{cost}_1(f, r') - \mathrm{cost}_2(f, r') \\ &= \mathrm{sel}(\mathrm{prev}(f, r'))\Delta(\mathrm{cost}(f) - \delta),\end{aligned}$$

where $\Delta = \mathrm{cache}(f, r) - \mathrm{cache}(f, \mathrm{prev}(r))$.

Based on the above formulation, we have

$$\mathrm{reduction}(r) = \sum\nolimits_{r' \neq r} \mathrm{contribution}(r', r).$$

Our second greedy strategy simply picks the rule r that maximizes $\mathrm{reduction}(r)$ as the next rule to be executed. Algorithm 6 presents the details of the idea. It is more costly than Algorithm 5 because update of $\mathrm{reduction}(r)$ at line 21 requires $O(n)$ rather than $O(1)$ time, where n is the number of rules.

Note that the computations of $\mathrm{cost}(r)$ and $\mathrm{reduction}(r)$ are still based on local decisions, namely, the *immediate* effect if a rule is executed. The actual effect, however, depends on the actual ordering of all rules and cannot be estimated accurately without finishing execution of all rules (or, enumerating all possible rule orders).

5.4.2 Discussion

If we only employ early exit without dynamic memoing, the optimal ordering problem remains NP-hard when the predicates/rules are correlated. However, we can have a greedy 4-approximation algorithm [1, 11]. The difference in this context is that the costs of the predicates no longer depend on the order of the rules. Rather, they are constants so approximation is easier. One might then wonder if combining early exit with precomputation (but not dynamic memoing) would make the problem even tractable, for now the costs of the predicates become the same (i.e., the lookup cost). Unfortunately, the problem remains NP-hard even for uniform costs when correlation is present [6].

Algorithm 6: A greedy algorithm based on expected overall cost reduction.

```
Input: $\mathcal{R} = \{r_1, ..., r_n\}$, a set of CNF rules
Output: $\mathcal{R}^\pi$, execution order of the rules
Let $Q$ be a priority queue $\langle(\text{reduction}(r), r)\rangle$ of the rules;
$\mathcal{R}^\pi \leftarrow \emptyset$;
foreach $r \in \mathcal{R}$ do
    Order $\text{predicate}(r)$ according to Lemma 3;
end
foreach $r \in \mathcal{R}$ do
    $\text{reduction}(r) \leftarrow 0$;
    foreach $r' \in \mathcal{R}$ such that $r' \neq r$ do
        foreach $f \in \text{feature}(r')$ do
            if $f \in \text{feature}(r)$ then
                $\text{reduction}(r) \leftarrow$
                  $\text{reduction}(r) + \text{contribution}(r', r, f)$;
            end
        end
    end
    Insert $(\text{reduction}(r), r)$ into $Q$;
end
while $Q$ is not empty do
    $r_{\max} \leftarrow \text{ExtractMax}(Q)$;
    Add $r_{\max}$ into $\mathcal{R}^\pi$;
    foreach $(\text{reduction}(r), r) \in Q$ do
        Update $\text{reduction}(r)$ by assuming that $r$ immediately
          follows $r_{\max}$;
    end
end
return $\mathcal{R}^\pi$;
```

5.4.3 *Optimization: Check Cache First*

We have proposed two greedy algorithms for ordering rules and predicates in each rule. The order is computed before running any rule and remains the same during matching. However, the greedy strategies we proposed are based on the "expected" rather than actual costs of the predicates. In practice, once we start evaluating the rules, it becomes clear that a feature is in the memo or not. One could then further consider dynamically adjusting the order of the remaining rules based on the current content of the memo. This incurs nontrivial overhead, though: we basically have to rerun the greedy algorithms each time we finish evaluating a rule. So in our current implementation we do not use this optimization. Nonetheless, we are able to reorder the predicates inside each rule at runtime based on the content of the memo. Specifically, we first evaluate predicates for which we have their features in the memo, and we still rely on Lemma 3 to order the remaining predicates.

5.5 Putting It All Together

The basic idea in this section is to order the rules such that we can decide on the output of the matching function with lowest computation cost for each pair. To order the rules we use a small random sample of the candidate pairs and estimate feature costs and selectivities for each predicate and rule. We then use Algorithm 5 or Algorithm 6 to order the rules. These two algorithms consider two different factors that affect the overall cost: 1) the expected cost of each rule, and 2) the expected overall cost reduction that executing this rule will have if the features computed for this rule are repeated in the following rules. We further evaluate the performance of both algorithms in our experiments.

6. INCREMENTAL MATCHING

So far we have discussed how to perform matching for a fixed set of fixed rules. We now turn to consider incremental matching in the context of an evolving set of rules.

6.1 Materialization Cost

To perform incremental matching, we materialize the following information during each iteration:

- For each pair: For each feature that was computed for this pair, we store the calculated score. Note that because we use lazy feature computation, we may not need to compute all feature values.
- For each rule: Store all pairs for which this rule is true.
- For each predicate: Store all pairs for which this predicate evaluated to false.

We show in our experiments that if we use straightforward techniques such as storing bitmaps of pairs that pass rules or predicates, the total memory needed to store this information for our data sets is less than 1GB.

6.2 Types of Matching Function Changes

An analyst often applies a single change to the matching function, re-runs EM, examines the output, then applies another change. We study different types of changes to the matching function and present our incremental matching algorithm for each type.

6.2.1 *Add a Predicate / Tighten a Predicate*

Algorithm 7: Add a predicate.

```
Input: $\mathcal{R}$, the set of CNF rules; $r$, the rule that was changed;
       $p$, the predicate added to $r$
Let $M(r)$ be the previously matched pairs by $r$;
Let $X$ be the unmatched pairs by $p$; $X \leftarrow \emptyset$;
foreach $c \in M(r)$ do
    if $p$ returns false for $c$ then
        $X \leftarrow X \cup \{c\}$;
    end
end
Let $\mathcal{R}'$ be the rules in $\mathcal{R}$ after $r$;
foreach $c \in X$ do
    Mark $c$ as an unmatch;
    foreach $r' \in \mathcal{R}'$ do
        if $r'$ returns true for $c$ then
            Mark $c$ as a match; break;
        end
    end
end
```

If a matching result contains pairs that should not actually match, the analyst can make the rules that matched such a pair more "strict" by either adding predicates, or making existing predicates more strict. For example, consider the following predicate

$$\text{Jaccard}(a.name, b.name) \geq 0.7.$$

We can make this more strict by changing it to

$$\text{Jaccard}(a.name, b.name) \geq 0.8.$$

In this case, we can obtain the new matching results incrementally by evaluating this modified predicate only for the pairs that were evaluated and matched by the rule we made stricter. Consider such a previously matched pair:

- If the modified predicate returns true, the pair is still matched.
- If the modified predicate returns false, the current rule no longer matches this pair. However, other rules in the matching function may match this pair, so we must evaluate the pair with the other rules until either a rule returns true or all rules return false.

We can use the same approach for adding a new predicate to a rule, because that can be viewed as making an empty predicate that always evaluates to true more strict. Algorithm 7 illustrates the procedure for adding a predicate.

6.2.2 Remove a Predicate / Relax a Predicate

Algorithm 8: Make a predicate less strict.

```
Input: $r$, the rule that was changed; $p$, the predicate of $r$ that
       was made less strict
Let $U(p)$ be the pairs for which $p$ returned false;
Let $Y$ be the pairs $p$ now returns true; $Y \leftarrow \emptyset$;
foreach $c \in U(p)$ and $c$ was an unmatch do
    if $p$ returns $\mathrm{true}$ for $c$ then
        $Y \leftarrow Y \cup \{c\}$;
    end
end
foreach $c \in Y$ do
    Mark $c$ as a match;
    foreach $p' \in \mathrm{predicate}(r)$ and $p' \neq p$ do
        if $p'$ returns $\mathrm{false}$ for $c$ then
            Mark $c$ as an unmatch; break;
        end
    end
end
```

In the case where pairs that should match are missing from the result, we might be able to fix the problem by either removing a predicate or making an existing predicate less strict. Consider again the predicate

$$\mathrm{Jaccard}(a.name, b.name) \geq 0.7.$$

We can make it less strict by changing it to

$$\mathrm{Jaccard}(a.name, b.name) \geq 0.6.$$

In both cases, all pairs for which this predicate returned false need to be re-evaluated. Consider such a previously unmatched pair:

- If the new predicate is false, the pair remains unmatched.
- If the new predicate is true, we will evaluate the other predicates in the rule.[2] If any of these predicates returns false, then the pair remains a non-match. Otherwise, this rule will return true for this pair, and it will be declared a match.

Algorithm 8 illustrates the details of the procedure for updating the matching result after making a predicate less strict. Removing a predicate follows similar logic and is omitted for brevity.

6.2.3 Remove a Rule

Algorithm 9: Remove a rule.

```
Input: $\mathcal{R}$, the set of CNF rules; $r$, the rule removed
Let $M(r)$ be the previously matched pairs by $r$;
Let $\mathcal{R}'$ be the rules in $\mathcal{R}$ after $r$;
foreach $c \in M(r)$ do
    Mark $c$ as an unmatch;
    foreach $r' \in \mathcal{R}'$ do
        if $r'$ returns $\mathrm{true}$ for $c$ then
            Mark $c$ as a match; break;
        end
    end
end
```

We may decide to remove a rule if it returns true for pairs that should not match. In such a case, we can re-evaluate the matching function for all pairs that were matched by this rule. Either another rule will declare this pair a match or the matching function will return false. Algorithm 9 illustrates this procedure.

6.2.4 Add a Rule

Algorithm 10: Add a rule.

```
Input: $\mathcal{R}$, the set of CNF rules; $r$, the rule added
Let $U(r)$ be the previously unmatched pairs by $\mathcal{R}$;
foreach $c \in U(r)$ do
    Mark $c$ as a match;
    foreach $p \in \mathrm{predicate}(r)$ do
        if $p$ returns $\mathrm{false}$ for $c$ then
            Mark $c$ as an unmatch; break;
        end
    end
end
```

One way to match pairs that are missed by a current matching function is to add a rule that returns true for them. In this case, inevitably, all non-matched pairs need to be evaluated by this rule. However, note that only the newly added rule will be evaluated for the non-matched pairs, which can be substantial savings over re-evaluating all rules. Algorithm 10 demonstrates this procedure.

7. EXPERIMENTAL EVALUATION

In this section we explore the impact of our techniques on the performance of various basic and incremental matching tasks. We ran experiments on a Linux machine with eight 2.80 GHz processors (each with 8 MB of cache) and 8 GB of main memory. We implemented our algorithms in Java. We used six real-world data sets as described below.

7.1 Datasets and Matching Functions

We evaluated our solutions on six real-world data sets. One data set was obtained from an industrial EM team. The remaining five data sets were created by students in a graduate-level class as part of their class project, where they had to crawl the Web to obtain, clean, and match data from two Web sites. Table 2 describes these six data sets. For ease of exposition, and due to space constraints, in the rest of this section we will describe experiments with the first (and largest) data set. Experiments with the remaining five data sets show similar results and are therefore omitted.

We obtained the Walmart/Amazon data set used in [7] from the authors of that paper. The dataset domain is electronics items from

[2] Note that, because we use the “check-cache-first” optimization, the order of the predicates within the rule is no longer fixed. In other words, different pairs may observe different orders. So we cannot just evaluate predicates that “follow” the changed one.

Data set	Source 1	Source 2	Table1 size	Table2 size	Candidate pairs	Rules	Used features	Total features
Products	Walmart	Amazon	2554	22074	291649	255	32	33
Restaurants	Yelp	Foursquare	3279	25376	24965	32	21	34
Books	Amazon	Barnes & Noble	3099	3560	28540	10	8	32
Breakfast	Walmart	Amazon	3669	4165	73297	59	14	18
Movies	Amazon	Bestbuy	5526	4373	17725	55	33	39
Video games	TheGamesDB	MobyGames	3742	6739	22697	34	23	32

Table 2: Real-world data sets used in the experiments.

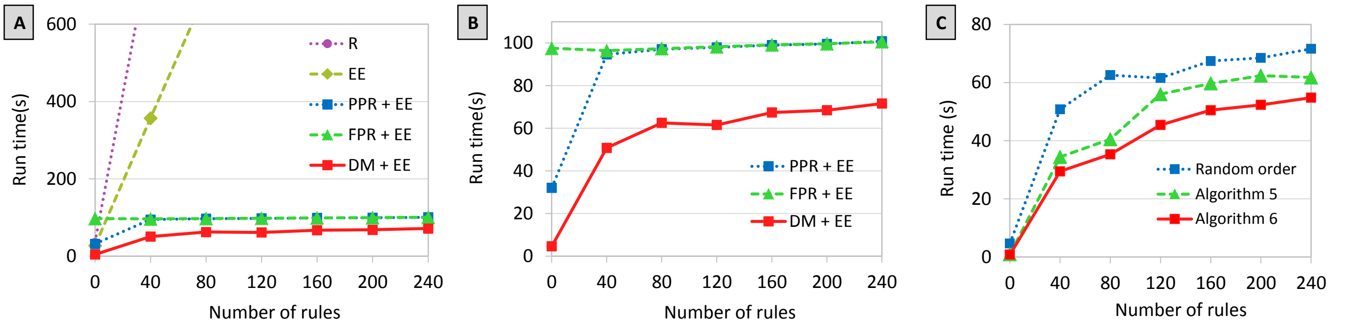


Figure 3: **(A)** Run time for different sizes of matching function for rudimentary baseline (R), early exit (EE), production precomputation baseline + early exit (PPR + EE), full precomputation baseline + early exit (FPR + EE), and dynamic memoing + early exit (DM + EE). **(B)** Zoom-in of A to compare methods that use precomputation/dynamic memoing. **(C)** Run time for different orderings of the set of rules/predicates: Random ordering, order by Algorithm 5, and order by Algorithm 6.

Function	Walmart	Amazon	µs
Exact Match	modelno	modelno	0.2
Jaro	modelno	modelno	0.5
Jaro Winkler	modelno	modelno	0.77
Levenshtein	modelno	modelno	1.22
Cosine	modelno	title	3.37
Trigram	modelno	modelno	4.79
Jaccard	modelno	title	6.75
Soundex	modelno	modelno	8.77
Jaccard	title	title	10.54
TF-IDF	modelno	title	12.18
TF-IDF	title	title	18.92
Soft TF-IDF	modelno	title	21.89
Soft TF-IDF	title	title	66.06

Table 3: Computation costs for features in the products data set

Walmart.com and Amazon.com. After the blocking step, we have $291,649$ candidate pairs. Gokhale et al. [7] have generated the ground truth for these pairs.

We generated 33 features using a variety of similarity functions based on heuristics that take into account the length and type of the attributes. Table 3 shows a subset of these features and their associated average computation times. The computation times of features vary widely.

Using a combination of manual and semi-automatic approaches, analysts from the EM team that originally created the data set have created a total of 255 matching rules. We will use this rule set as a basis from which to create and evaluate a variety of matching functions. Figure 4 shows two sample rules for this data set.

$R1$ Jaro Winkler(m, m) $\geq 0.97 \wedge$ Jaro(m, m) ≥ 0.95
$\wedge$ Soft TF-IDF(m, t) < 0.28
$\wedge$ TF-IDF(m, t) $< 0.25 \wedge$ Cosine(t, t) ≥ 0.69

$R2$ Jaccard(t, t) $< 0.4 \wedge$ TF-IDF(t, t) < 0.55
$\wedge$ Soft TF-IDF(t, t) $\geq 0.63 \wedge$ Jaccard ≥ 0.34
$\wedge$ Levenshtein(m, m) < 0.72
$\wedge$ Jaro Winkler(m, m) < 0.05

Figure 4: Sample rules extracted from the random forest. m, t stand for modelno and title respectively.

7.2 Early Exit + Dynamic Memoing

Figure 3A shows the effect of early exit and precomputing (memoing) feature values on matching time as we use an increasingly larger rule set. For example, to generate the data point corresponding to 20 rules, we randomly selected 20 rules and measured the time to apply them to the data set. For each data point we report the average running time over three such random sets of rules.

We compare the run time for baseline, early exit, production precomputation + early exit, full precomputation + early exit, and dynamic memoing + early exit. For production precomputation, which we described as one of our baselines in Section 4.1.2, we assume that we know all the features used in the rules. We call this "production precomputation" because it is feasible only if the set of rules for matching is already finalized. In full precomputation, we know a superset of features that the analyst will choose from to make the rule set. In such a case, we may precompute values for features that will never be used. We compare these approaches with dynamic memoing + early exit proposed in this paper.

We can see that the rudimentary baseline has a very steep slope, and around 20 rules, it takes more than 10 minutes to complete. The early exit curve shows significant improvement over baseline, however, it is still slow compared to either the precomputation baseline or early exit with dynamic memoing. Figure 3B zooms in and shows the curves for the full and production precomputation baselines and dynamic memoing. We can see that using dynamic memoing + early exit can significantly reduce matching time.

In this subsection, we have not considered the optimal ordering problem, and we ran dynamic memoing with a random ordering of the rules and predicates in each rule. In the next subsection, we further study the effectiveness of our greedy algorithms on optimizing orderings of predicates/rules.

7.3 Optimal Ordering

Figure 3C shows runtime as we increase the number of rules for "dynamic memoing + early exit" with random ordering of predicates/rules, as well as that with orderings produced by the two greedy strategies presented in Algorithm 5 and Algorithm 6. Each data point was generated using the same approach described in the previous subsection. We used a random sample consisting of 1% of the candidate pairs for estimating feature costs and predicate selectivities. We can see that the orderings produced by both of these algorithms are superior to the random ordering.

We further observe that Algorithm 6 is faster than Algorithm 5, perhaps due to the fact that its decision is based on a global optimization metric that considers the overall cost reduction by placing a rule before other rules. As the number of rules increases, the impact is less significant, because most of the features have to be computed. Nonetheless, matching using Algorithm 6 is still faster even when we use 240 rules in the matching function.

7.4 Memory Consumption

We store the similarity values in a two dimensional array. We assign each pair an index based on their order in the input table. Similarly, we assign each feature a random order and an index based on the order. In the case of the precomputation baseline, this memo is completely filled with feature values before we start matching. In the case of dynamic memoing, we fill in the memo as we run matching and the analyst makes changes to the rule set. Therefore, the memory consumption of both approaches is the same. For this dataset, if we use all rules, the two-dimensional array takes 22 MB of space, which includes the space for storing the actual floats as well as the bookkeeping overhead for the array in Java. For incremental matching, we store a bitmap for each rule as well as for each predicate. In our implementation, we use a boolean array for each bitmap. For this dataset, we have 255 rules and a total of $1,688$ predicates. These bitmaps occupy 542 MB.

For our dataset, the two-dimensional array and bitmaps fit comfortably in memory. For a data set where this is not true, one could consider avoiding an array and using a hash-map for storing similarity values. Since we do not compute all the feature values, this would lead to less memory consumption, although the lookup cost for hash-maps would be more expensive.

7.5 Cost Modeling and Analysis

To illustrate accuracy of our cost models, in Figure 5A we compare the actual run time of "dynamic memoing + early exit" versus run time estimated by the cost model for random ordering of rules as well as rules ordered by Algorithm 6. The two curves follow each other closely.

To compute the selectivity of each predicate, we select a sample of the candidate pairs, evaluate each predicate for the pairs in the sample and compute the percentage of pairs that pass each predicate. In our experiments, we observed that using a 1% sample can give relatively accurate estimates of the selectivity, and increasing the sample size did not change the rule ordering in a major way. We used the same small sample approach to estimate feature costs.

Figure 5B shows the actual matching time when we use all the rules for the data set as we increase number of pairs. As we assumed in our cost modeling, the matching cost increases linearly as we increase number of pairs. Given this increase proportional to the number of pairs (which is itself quadratic in the number of input records), the importance of performance enhancing techniques to achieve interactive response times increases with larger data sets.

7.6 Incremental Entity Matching

Our first experiment examines the "add rule" change. Adding a rule can be expensive for incremental entity matching because we need to evaluate the newly added rule for all the unmatched pairs.

To test how incremental matching performs for adding a new rule, we conducted the following experiment. We start from an empty matching function without any rules. We then add the first rule to the matching function, run matching with this single-rule matching function, and materialize results. Next, we add the second rule and measure the time required for incremental matching. In general, we run matching based on k rules, and then run incremental matching for the $(k+1)$-th rule when it is added. We repeat this for $1 \leq k \leq 240$.

We consider two variations of incremental algorithm. In the *precomputation* variation, all the rules in the matching function are evaluated. Note that we use early exit and the optimization discussed in Section 5.4.3 with this variation to reduce unnecessary lookups. The second variation is *fully incremental*. In this case we not only lookup the stored feature values, but also only evaluate part of the matching function for the subset of candidate pairs that will be affected by this operation. In particular, for the "add rule" operation, all the non-matched pairs need to be evaluated by just the new rule that is added, and all the rules in the matching function do not need to be evaluated.

Figure 5C shows the results for the add-rule experiment. We can see that in the first iteration, both variations are slow. This is because there is no materialized result to use (i.e. the memo is empty). However, from the second iteration onwards we can see that the cost of the precomputation baseline steadily increases whereas the cost of fully incremental is mostly constant and significantly smaller than that of the precomputation baseline. This is because the precomputation baseline performs unnecessary lookups and evaluates all the rules in the matching function. The incremental approach just evaluates the newly added rule and thus it does not slow down as the number of rules increases.

In certain runs both of the variations experience a sudden increase in the running time. These are the cases in which the new rule requires many feature computations, because either there was a new feature, or the feature was not in the memo, and this feature was "reached" in the rule evaluation (it might not be reached, for example, if a predicate preceding the feature evaluates to false.)

Figure 6 shows run time of incremental EM for different changes to the matching function. To illustrate how the numbers were generated, assume that we want to measure the incremental run time for adding a predicate. We randomly selected 100 predicates, removed the predicate, ran EM and materialized the results, then added the predicate to the rule, and measured the run time. The rest of the numbers in the table were generated in a similar manner.

For tightening the thresholds, we randomly selected a predicate, and for that predicate we randomly chose one of the values in

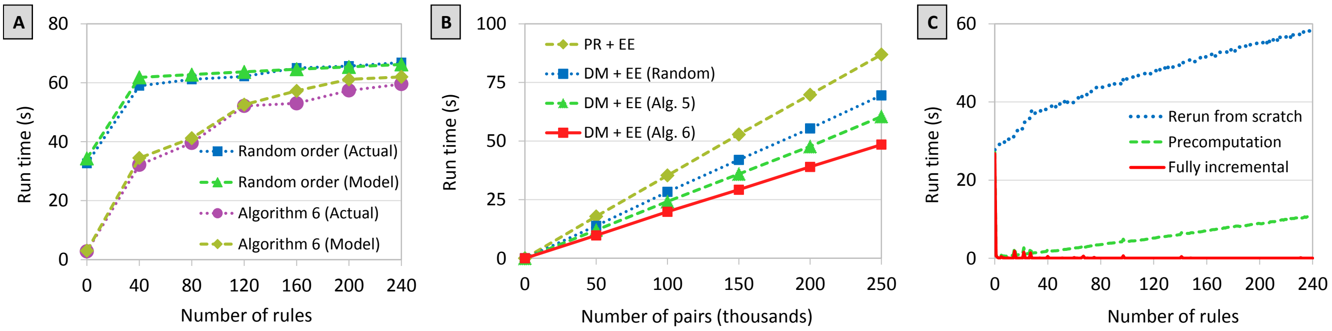


Figure 5: **(A)** Actual run time versus run time estimated by the cost model for random ordering of rules and rules ordered by Algorithm 6. **(B)** Run time as we increase number of pairs for production precomputation + early exit (PPR + EE), and dynamic memoing + early exit (DM + EE) for random ordering of rules, rules ordered by Algorithm 5, and ordered by Algorithm 6. **(C)** Run time with dynamic memoing + early exit as we add rules one by one to the matching function in three cases: 1) Rerun matching from scratch, 2) Precomputation: lookup memoed feature values but evaluate all rules 3) Fully incremental: lookup memoed feature values but only evaluate the newly added rule.

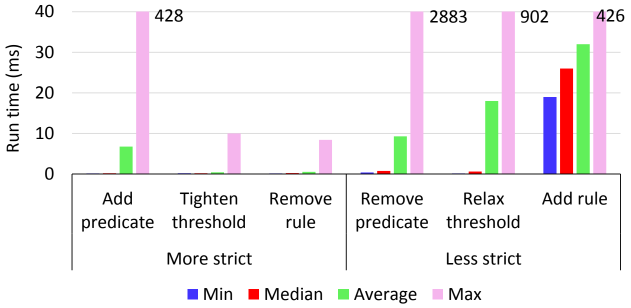


Figure 6: Incremental EM run time for different changes to the matching function that make it more/less strict.

$\{0.1, 0.2, 0.3, 0.4, 0.5\}$ that could be applied to the predicate. For example, assume that the predicate is $\text{Jaccard}(a.name, b.name) \geq 0.6$. To tighten the threshold, we add a random value to the threshold from $\{0.1, 0.2, 0.3, 0.4\}$, because adding 0.5 makes the threshold larger than 1. If the predicate uses a $\geq$ operation we add the value to the current threshold, and if it uses a $\leq$ operation we subtract the value from the current threshold. The procedure is similar for relaxing thresholds.

We can see that making the matching function more strict by adding a predicate, tightening the threshold, and removing a rule on average takes no more than about 6 milliseconds. On the other hand, making the function less strict could take up to 34 milliseconds on average. This cost is due to the fact that we may need to calculate new features for a fraction of candidate pairs.

8. CONCLUSIONS

We have focused on scenarios where an analyst iteratively designs a set of rules for an EM task, with the goal of making this process as interactive as possible. Our experiments with six real-world data sets indicate that "memoing" the results of expensive similarity functions is perhaps the single most important factor in achieving this goal, followed closely by the implementation of "early-exit" techniques that stop evaluation as soon as a matching decision is determined for a given candidate pair.

In the context of rule creation and modification it may not be desirable or even possible to fully precompute similarity function results in advance. Our just-in-time "memoing" approach solves this problem, dynamically storing these results as needed; however, the interaction of the on-demand memoing and early-exit evaluation creates a novel rule and predicate ordering optimization problem. Our heuristic algorithms to solve this problem provide significant further reductions in running times over more naive approaches.

Finally, in the context of incremental rule iterative development, we show that substantial improvements in running times are possible by remembering the results of previous iterations and on the current iteration only computing the minimal delta required by a given change.

From a broader perspective, this work joins a small but growing body of literature which asserts that for matching tasks, there is often a "human analyst in the loop," and rather than trying to remove that human, attempts to make him more productive. Much room for future work exists in integrating the techniques presented here with a full system and experimenting with its impact on the analyst.

9. REFERENCES

[1] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD Conference*, 2004.

[2] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *The International Journal on Very Large Data Bases*, 18(1):255–276, 2009.

[3] P. Christen. *Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection*. Springer, 2012.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[5] A. Doan, A. Halevy, and Z. Ives. *Principles of data integration*. Morgan Kaufmann, 2012.

[6] U. Feige, L. Lovász, and P. Tetali. Approximating min-sum set cover. In *APPROX*, 2002.

[7] 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.

[8] J. M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Trans. Database Syst.*, 23(2):113–157, 1998.

[9] L. Kolb, A. Thor, and E. Rahm. Dedoop: efficient deduplication with hadoop. *VLDB*, 2012.

[10] P. Konda, S. Das, P. Suganthan GC, A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. Naughton, et al. Magellan: Toward building entity matching management systems. *VLDB*, 2016.

[11] K. Munagala, S. Babu, R. Motwani, and J. Widom. The pipelined set cover problem. In *ICDT*, 2005.

[12] J. Nielsen. *Usability engineering*. Academic Press, 1993.

[13] P. Suganthan et al. Why big data industrial systems need rules and what we can do about it. In *SIGMOD Conference*, 2015.

[14] J. Wang, G. Li, J. X. Yu, and J. Feng. Entity matching: How similar is similar. *VLDB*, 2011.

[15] S. E. Whang and H. Garcia-Molina. Incremental entity resolution on rules and data. *The International Journal on Very Large Data Bases*, 23(1):77–102, 2014.

Convergent Interactive Inference with Leaky Joins

Ying Yang
University at Buffalo
yyang25@buffalo.edu

Oliver Kennedy
University at Buffalo
okennedy@buffalo.edu

ABSTRACT

One of the primary challenges in graphical models is inference, or re-constructing a marginal probability from the graphical model's factorized representation. While tractable for some graphs, the cost of inference grows exponentially with the graphical model's complexity, necessitating approximation for more complex graphs. For interactive applications, latency is the dominant concern, making approximate inference the only feasible option. Unfortunately, approximate inference can be wasteful for interactive applications, as exact inference can still converge faster, even for moderately complex inference problems. In this paper, we propose a new family of convergent inference algorithms (CIAs) that bridge the gap between approximations and exact solutions, providing early, incrementally improving approximations that become exact after a finite period of time. We describe two specific CIAs based on a cryptographic technique called linear congruential generators, including a novel incremental join algorithm for dense relations called Leaky Joins. We conclude with experiments that demonstrate the utility of Leaky Joins for convergent inference: On both synthetic and real-world probabilistic graphical models, Leaky Joins converge to exact marginal probabilities almost as fast as state of the art *exact* inference algorithms, while simultaneously achieving approximations that are almost as good as state of the art *approximation* algorithms.

1. INTRODUCTION

Probabilistic graphical models (PGMs) are a factorized encoding of joint (multivariate) probability distributions. Even large distributions can often be compactly represented as a PGM. A common operation on PGMs is inference, or reconstructing the marginal probability for a subset of the variables in the full joint distribution. Existing inference algorithms are either exact or approximate. Exact algorithms [9] like variable elimination and belief propagation produce exact results, but can be slow. On the other hand, approximate algorithms [44,45] like Gibbs sampling generate estimates within any fixed time bounds, but only converge asymptotically to exact results.

Over the past decade, a class of model database systems have begun to add support for probabilistic graphical models (PGMs) within database engines, allowing graphical models to be queried through SQL [13,36,42], combined with other data for joint analysis [20, 22], or used for analytics over messy data [29,40,41].

Model database systems typically employ approximate inference techniques, as model complexity can vary widely with different usage patterns and responsiveness is typically more important than exact results. However, exact inference can sometimes produce an exact result *faster* than it takes an approximate algorithm to converge, even for moderately complex inference problems. Furthermore, in interactive settings, the user may be willing to wait for more accurate results. In either case, the choice of whether or not use an exact algorithm must wait until the system has already obtained an approximation.

In this paper, we explore a family of *convergent inference algorithms* (CIAs) that simultaneously act as both approximate and exact inference algorithms: Given a fixed time bound, a CIA can produce a bounded approximate inference result, but will also terminate early if it is possible to converge to an exact result. Like a file copy progress bar, CIAs can provide a "result accuracy progress bar" that is guaranteed to complete eventually. Similar to online-aggregation [18] (OLA), CIAs give users and client applications more control over accuracy/time trade-offs and do not require an upfront commitment to either approximate or exact inference.

We propose two specific CIAs that use the relationship between inference and select-join-aggregate queries to build on database techniques for OLA [18]. Our algorithms specialize OLA to two unique requirements of graphical inference: dense data and wide joins. In classical group-by aggregate queries, the joint domain of the group-by attributes is sparse: Tables in a typical database only have a small portion of their active domain populated. Furthermore, classical database engines are optimized for queries involving a small number of large input tables. Conversely, in graphical inference, each "table" is small and dense and there are usually a large number of tables with a much more complicated join graph.

The density of the input tables (and by extension, all intermediate relations) makes it practical to sample directly from the output of a join, since each sample from the active domain of the output relation is likely to hit a row that is

Series ISSN: 2367-2005

10.5441/002/edbt.2017.33

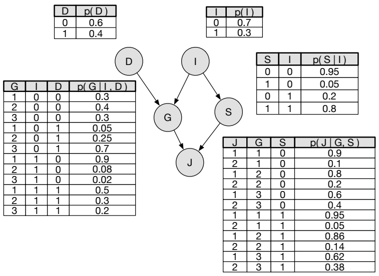
present and non-zero. Hence, our first, naive CIA samples directly from the output of the select-join component of the inference query, using the resulting samples to predict the aggregate query result. To ensure convergence, we leverage a class of pseudorandom number generators called Linear Congruential Generators [31, 34] (LCGs). The cyclicity of LCGs has been previously used for coordinating distributed simulations [6]. Here, we use them to perform random sampling from join outputs *without* replacement, allowing us to efficiently iterate through the join outputs in a shuffled order. These samples produce bounded-error estimates of aggregate values. After the LCG completes one full cycle, every row of the join has been sampled exactly once and the result is exact.

Unfortunately, the domain of the join output for an inference query can be quite large and this naive approach converges slowly. To improve convergence rates, we propose a new online join algorithm called *Leaky Joins* that produces samples of a query's result in the course of normally evaluating the query. Systems for relational OLA (e.g., [15, 18, 21]) frequently assume that memory is the bottleneck. Instead, Leaky Joins are optimized for small input tables that make inference more frequently compute-bound than IO-bound. Furthermore, the density of the input (and intermediate) tables makes it possible to use predictable, deterministic addressing schemes. As a result, Leaky Joins can obtain unbiased samples efficiently without needing to assume a lack of correlation between attributes in the input.

The Leaky Joins algorithm starts with a classical bushy query plan. Joins are evaluated in parallel, essentially *"leaking"* estimates for intermediate aggregated values — marginal probabilities in our motivating use case — from one intermediate table to the next. One full cycle through a LCG is guaranteed to produce an exact result for joins with exact inputs available. Thus, initially only the intermediate tables closest to the leaves can produce exact results. As sampling on these tables completes a full cycle, they are marked as stable, sampling on them stops, and the tier above them is permitted to converge. In addition to guaranteeing convergence of the final result, we are also able to provide confidence bounds on the approximate results prior to convergence. As we show in our experiments, the algorithm satisfies desiderata for a useful convergent-inference algorithm: *computation performance* competitive with exact inference on simple graphs, and *progressive accuracy* competitive with approximate inference on complex graphs.

Our main motivation is to generate a new type of inference algorithm for graphical inference in databases. Nevertheless, we observe that Leaky Joins can be adapted to any aggregate queries over small but dense tables.

Specifically, our contributions include: (1) We propose a new family of Convergent Inference Algorithms (CIAs) that provide approximate results over the course of inference, but eventually converge to an exact inference result, (2) We cast the problem of Convergent Inference as a specialization of Online Aggregation, and propose a naive, *constant-space* convergent inference algorithm based on Linear Congruential Generators, (3) We propose Leaky Joins, a novel Online Aggregation algorithm specifically designed for Convergent Inference, (4) We show that Leaky Joins have time complexity that is no more than one polynomial order worse than classic exact inference algorithms, and provide an $\epsilon-\delta$ bound to demonstrate that the approximation accuracy is competitive with common approximation techniques, (5) We present experimental results on both synthetic and real-world graph data to demonstrate that (a) Leaky Joins gracefully degrade from exact inference to approximate inference as graph complexity rises. (b) Leaky Joins have exact inference costs competitive with classic exact inference algorithms, and approximation performance competitive with common sampling techniques, (6) We discuss lessons learned in our attempts to design a convergent inference algorithm using state-of-the-art incremental view maintenance systems [4, 25].



D	p(D)
0	0.6
1	0.4

I	p(I)
0	0.7
1	0.3

S	I	p(S | I)
0	0	0.95
1	0	0.05
0	1	0.2
1	1	0.8

G	I	D	p(G | I , D)
1	0	0	0.3
2	0	0	0.4
3	0	0	0.3
1	0	1	0.05
2	0	1	0.25
3	0	1	0.7
1	1	0	0.9
2	1	0	0.08
3	1	0	0.02
1	1	1	0.5
2	1	1	0.3
3	1	1	0.2

J	G	S	p(J | G, S)
1	1	0	0.9
2	1	0	0.1
1	2	0	0.8
2	2	0	0.2
1	3	0	0.6
2	3	0	0.4
1	1	1	0.95
2	1	1	0.05
1	2	1	0.86
2	2	1	0.14
1	3	1	0.62
2	3	1	0.38

Figure 1: A simple Student Bayesian network

2. BACKGROUND AND RELATED WORK

In this section, we introduce notational conventions that we use throughout the paper and briefly overview probabilistic graphical models, inference and on-line aggregation.

2.1 Bayesian Networks

Complex systems can often be characterized by multiple interrelated properties. For example, in a medical diagnostics system, a patient might have properties including symptoms, diagnostic test results, and personal habits or predispositions for some diseases. These properties can be expressed for each patient as a set of interrelated random variables. We write sets of random variables in bold (e.g., $\mathbf{X} = \{X_i\}$). Denote by $p(\mathbf{X})$ the probability distribution of $X_i \in \mathbf{X}$ and by $p(x)$ the probability measure of the event $\{X_i = x\}$. Let $\mathbf{X}\backslash\mathbf{Y}$ denote the set of variables that belong to $\mathbf{X}$ but do not belong to $\mathbf{Y}$.

A Bayesian network (BN)[1] represents a joint probability distribution over a set of variables $\mathbf{X}$ as a directed acyclic graph. Each node of the graph represents a random variable X_i in $\mathbf{X}$. The parents of X_i are denoted by $pa(X_i)$, the children of X_i are denoted by $ch(X_i)$.

A Bayesian network compactly encodes a joint probability distribution using the Markov condition: Given a variable's parents, the variable is independent of all of its non-descendants in the graph. Thus, the full joint distribution is given as:

$$P(\mathbf{X}) = \prod_i P(X_i|pa(X_i))$$

Every random variable X_i is associated with a conditional probability distribution $P(X_i|pa(X_i))$. The joint probability distribution is factorized into a set of $P(X_i|pa(X_i))$ called factors denoted by ϕ_i or factor tables if X_i is discrete. Denote by $scope(\phi_i)$ the variables in a factor ϕ_i. Finally, we use $attrs(\phi_i) = scope(\phi_i) \cup \{p_{\phi_i}\}$ to denote the attributes

[1] Although our focus here is inference on directed graphical models (i.e. Bayesian networks), the same techniques can be easily adapted for inference in undirected graphical models.

of the corresponding factor table: the variables in the factor's scope and the probability of a given assignment to X_i given fixed assignments for its parents. A full BN can then be expressed as the 2-tuple $\mathcal{B} = (\mathcal{G}(\mathbf{X}), \Phi)$, consisting of the graph and the set of all factors.

Example 1. *Consider four random variables **I**ntelligence, **D**ifficulty, **G**rade, **S**AT, and **J**ob in a Student Bayesian network. The four variables I, D, S, J have two possible values, while G has 3. A relation with $2 \cdot 2 \cdot 2 \cdot 2 \cdot 3 = 48$ rows is needed to represent this joint probability distribution. Through the Markov condition, the graph can be factorized into the smaller Bayesian network given in Figure 1. For a graph with a large number of variables with large domains, factorization can reduce the size significantly.*

2.2 Inference

Inference in BNs usually involves computing the posterior marginal for a set of query variables $\mathbf{X}_q$ given a set of evidence, denoted by E. For example, $E = \{X_1 = x_1, X_3 = x_3\}$ fixes the values of variables X_1 and X_3. Denote by $\mathbf{X}_E$ the set of observed variables (e.g., $\mathbf{X}_E = \{X_1, X_3\}$). The posterior probability of $\mathbf{X}_q$ given E is

$$P(\mathbf{X}_q|E) = \frac{P(\mathbf{X}_q, E)}{P(E)} = \frac{\sum_{\mathbf{X}\setminus\{\mathbf{X}_q, \mathbf{X}_E\}} P(\mathbf{X})}{\sum_{\mathbf{X}\setminus\mathbf{X}_E} P(\mathbf{X})}.$$

The marginalization of $X_1 \ldots X_i$ over a joint probability distribution is equivalent to a select-join-aggregate query computed over the ancestors of $X_1 \ldots X_i$:

```
SELECT X_1,...,X_i,
           SUM(p_1 * ... * p_N) AS prob
FROM factor_1 NATURAL JOIN ...
                NATURAL JOIN factor_N
WHERE E_1 = e_1 AND ... AND E_k = e_k
GROUP BY X_1,...,X_i;
```

Applying evidence to a graphical model is computationally straightforward and produces a strictly simpler graphical model. As a result, without loss of generality, we ignore evidence and focus exclusively on straightforward inference queries of the form P($\mathbf{X}_q$).

2.2.1 Exact Inference

Variable Elimination. Variable elimination mirrors aggregation push-down [10], a common query optimization technique. The idea is to avoid the exponential blowup in the size of intermediate, joint distribution tables by pushing aggregation down through joins over individual factor tables. As in query optimization, join ordering plays a crucial role in variable elimination, as inference queries often have join graphs with high hypertree width. Intermediate materialized aggregates in VE are typically called separators (denoted S), intermediate (materialized) joins are called cliques (C), variables aggregated away between a clique and the following separator are called clique variables (denoted $var(C)$), and their inputs are called clique factors.

Example 2. *The marginal probability distribution of J in Figure 1 can be expressed by $p(J) = \sum_{D,I,S,G} p(D, I, S, G, J)$. We choose to first marginalize out D by constructing C_D's separator S_D:*

$$S_D[G, I] = \sum_D C_D[D, G, I] = \sum_D \phi_D[D] \bowtie_D \phi_G[D, G, I]$$

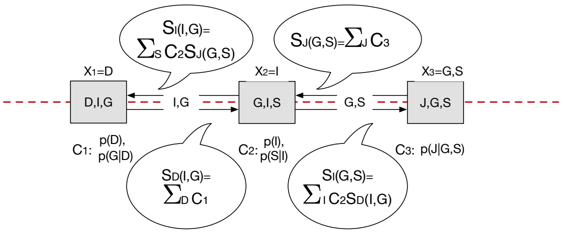


Figure 2: Clique tree for Student BN graph in Figure 1

Next, we marginalize out I by computing

$$S_I[G, S] = \sum_I C_I[G, I, S] = \sum_I \boldsymbol{S}_D[G, I] \bowtie_I \phi_I[I] \bowtie \phi_S[I, S]$$

The marginalization of G and S follows a similar pattern, leaving us with $C_q = S_S[J] = p(J)$.

The limiting factor in the computational cost of obtaining a separator is enumerating the rows of the clique. Assuming that the distribution over each variable X_i has N possible outcomes ($|dom(X_i)| = N$), the cost of computing separator S with clique C will be $O(N^{|scope(C)|})$. Tree-width in graphical models (related to query hypertree width) is the size of the largest clique's scope ($\max_C(|scope(C)|)$), making variable elimination exponential-cost in the graph's tree-width.
Belief Propagation. Belief propagation generalizes variable elimination by allowing information to flow in both directions along the graph, messages are sent along each cluster's separator by summing out all uncommon variables between the two clusters. The process creates, for each variable in the graph, its full conditional probability given all other variables in the graph. Figure 2 shows the message passing process for the graph in Figure 1. Although belief propagation is more efficient for performing multiple simultaneous inference operations in parallel, for singleton tasks it is a factor of two slower than variable elimination. Thus, in this paper we use variable elimination as a representative example of exact inference.

2.2.2 Approximate Inference

Markov Chain Monte Carlo Inference. MCMC is a family of sampling techniques that generate sequences of samples. Intuitively, the first element in the sequence is drawn from the prior and successive samples are drawn from distributions that get increasingly closer to the posterior. For example, we might draw one assignment of values in $\mathbf{X}$ with each variable X_i following the conditional probability distribution $p(X_i|pa(X_i))$ in the topological order of the graph. Then, we iteratively re-sample one variable's value at a time according to its factor table, given the current assignments for its parents and children. The longer we continue re-sampling, the less the sample is biased by its initial value. We use **Gibbs sampling** as a representative MCMC inference algorithm.
Loopy Belief Propagation. Loopy belief propagation is the same as belief propagation, but operates on a loopy cluster graph instead of a clique tree. This change makes the cluster smaller than those in clique tree and makes message passing steps less expensive. There is a trade-off between cost and accuracy in loopy-belief propagation, as join graphs that allow fast propagation may create a poor approximation of the result. This tradeoff is antithetical to our goal

of minimizing heuristic tradeoffs, making loopy-belief propagation a poor fit for our target applications. As a result we focus on Gibbs sampling as a representative example of approximate inference.

2.3 Online Aggregation

Starting with work by Hellerstein et.al. [18], Olken [30], and others, a large body of literature has been developed for so-called online aggregation (OLA) or approximate query processing (AQP) systems. Such systems replace pipeline-blocking operators like join and aggregate with sampling-based operators that permit approximate or partial results to be produced immediately, and iteratively refined the longer the user is willing to wait. The work most closely related to our own efforts is on OLA [16, 17, 18]. OLA systems use query evaluation strategies that estimate and iteratively refines the output of aggregate-joins. Given enough time, in most systems, the evaluation strategy eventually converges to a correct result. As in random sampling, $\epsilon - \delta$ bounds can be obtained, for example using Hoeffding's inequality. A key challenge arising in OLA is how to efficiently generate samples of source data. Sampling without replacement allows the algorithm to converge to the correct result once all samples have been exhausted, but has high space requirements, as it is necessary to keep track of the sampling order. Conversely, sampling with replacement is not guaranteed to ever converge to the correct answer. One of our key contributions in this paper is a specialization of OLA to graphical models called Cyclic Sampling, which permits sampling without replacement using only constant-space. Numerous other systems have since adapted and improved on the idea of OLA. Aqua [1] uses key constraints for improved stratified sampling. BlinkDB [2, 3] and Derby [23] maintain prematerialized stratified samples to rapidly answer approximate queries. GLADE [33] and DBO [15, 21] exploit file buffering to opportunistically generate samples of a query result in the course of normal query evaluation.

3. CONVERGENT INFERENCE

Running-time for variable elimination $O(N^{|scope(C)|})$, is dominated by tree-width, and strongly depends on the elimination ordering (already an NP-hard problem [26]). Since the running time grows exponentially in the size of largest clique cluster C_{max}, the running complexity can have high variance depending on the order. Because the cost is exponential, even a small increase in complexity can change the runtime of variable elimination from seconds to hours, or even days. In short, predicting whether an exact solution is feasible is hard enough that most systems simply rely exclusively on approximation algorithms. On other hand, approximate inference may get asymptotically close to an answer, but it will never fully converge. Thus, most applications that benefit from exact results must rely on either human intuition to decide.

The goal and first contribution of this paper is to introduce *convergent inference*, a specialized form of approximate inference algorithm that is guaranteed to eventually converge to an exact result. In this section, we develop the idea of convergent inference and propose several convergent inference algorithms, or CIAs. A CIA eventually produces an exact result, but can be interrupted at any time to quickly produce a bounded approximation. More precisely, a CIA should satisfy the following conditions: (1) After a fixed period of time t, a CIA can provide approximate results with $\epsilon - \delta$ error bounds, such that $P\left(|P_t - P_{exact}| < \epsilon\right) > 1 - \delta$; and (2) A CIA can will obtain the exact result P_{exact} in a bounded time t_{exact}.

Ideally, we would also like a CIA to satisfy two additional conditions: (3) The time complexity required by a good CIA to obtain an exact result should be competitive with variable elimination; and (4) The quality of the approximation produced by a good CIA should be competitive with the approximation produced by a strictly approximate inference algorithm given the same amount of time.

We first introduce a fundamental algorithm called cyclic sampling which performs pseudo-random sampling without replacement from the joint probability distribution of the graph. This algorithm is guaranteed to converge, but requires an exponential number of samples to do so. We then present an improved CIA based on classical aggregate-join query processing that relies on a novel "leaky join" operator. Finally, we discuss lessons learned in a failed attempt to combine cyclic sampling with state-of-the-art techniques for incremental view maintenance.

All three of our approaches draw on the relationship between graphical inference and aggregate query processing. However, though the problems are similar, we re-emphasize that there are several ways in which graphical inference queries violate assumptions made in classical database query-processing settings. First, conditional probability distributions are frequently dense, resulting in many-many relationships on join attributes. Correlations between variables are also common, so graphical models often have high tree widths. By comparison, the common case for join and aggregation queries is join graphs with a far smaller number of tables, simpler (e.g., foreign key) predicates, and typically low tree widths.

Finally, we note that although we use graphical models as a driving application, similar violations occur in other database applications (e.g., scientific databases [39]). The algorithms we present could be adapted for use in these settings as well.

3.1 Cyclic Sampling

We first discuss a naive form of convergent inference called cyclic sampling that forms the basis for each of our approaches. Recall that each intermediate table (the separator) is an aggregate computed over a join of factor tables (the clique), and that the domain of the clique is (or is very nearly) a cartesian product of the attributes in its scope. In principle, one could compute an entire separator table by scanning over the rows of its clique.

We note that input factor tables and the separator tables are typically small enough to remain in memory. Thus, using array-indexed storage for the clique tables is feasible and the cost of accessing one row of the clique is a constant. Consequently, efficient random sampling on the joint probability distribution is possible, and the marginal probabilities of interest can be incrementally approximated as in OLA [18].

The key insight of cyclic sampling is that if this random sampling is performed without replacement, it will eventually converge to an exact result if we reach a point where each row of the clique has been sampled exactly once. Unfortunately, sampling without replacement typically has space complexity linear in the number of items to be sampled, which is exponential in the number of variables. Fortunately for graphical inference, a there exists a class of so-called cyclic pseudorandom number generators that iteratively construct pseudorandom sequences of non-repeating integers in constant space.

A cyclic pseudorandom number generator generates a sequence of non-repeating numbers in the range $[0, m)$ for some given period m with members that exhibit minimal pairwise correlation. We use Linear Congruential Generators (LCGs) [31, 34], which generate a sequence of semi-random numbers with a discontinuous piecewise linear equation defined by the recurrence relation:

$$X_n = (aX_{n-1} + b) \mod m \tag{1}$$

Here X_n is the nth number of the sequence, and X_{n-1} is the previous number of the sequence. The variables a, b and m are constants: a is called the multiplier, b the increment, and m the modulus. The key, or seed, is the value of X_0, selected uniformly at random between 0 and m. In general, a LCG has a period no greater than m. However, if a, b, and m are properly chosen, then the generator will have a period of exactly m. This is referred to as a maximal period generator, and there have been several approaches to choosing constants for maximal period generators [24,28]. In our system, we follow the Hull-Dobell Theorem [38], which states that an LCG will be maximal if

1. m and the offset b are relatively prime.
2. $a - 1$ is divisible by all prime factors of m.
3. $a - 1$ is divisible by 4 if m is divisible by 4.

LCGs are fast and require only constant memory. With a proper choice of parameters a, b and m, a LCG can produce maximal period generators and pass formal tests for randomness. Parameter selection is outlined in Algorithm 1.

Algorithm 1 `InitLCG(totalSamples)`

```
Require: $totalSamples$: the total number of samples
Ensure: $a$, $b$, $m$: LCG Parameters
 1: $m \leftarrow totalSamples$
 2: $S \leftarrow$ prime factors of $m$
 3: for each $s$ in $S$ do
 4:     $a \leftarrow a \times s$
 5: if $m = 0 \mod 4$ and $a \neq 0 \mod 4$ then
 6:     $a \leftarrow 4a + 1$
 7: $b \leftarrow$ any coprime of $m$ smaller than $m$
```

The cyclic sampling process itself is shown in Algorithm 2. Given a Bayesian network $\mathcal{B}$=($\mathcal{G}(\mathbf{X})$,Φ), and a marginal probability query Q = $p(\mathbf{X}_q)$, we first construct the LCG sampling parameters (lines 1-5): a, b and m according to Hull-Dobell Theorem for the total number of samples in the joint probability distribution p($\mathbf{X}$). The sampling process (starts at Line 16) constructs an index by obtaining the next value from the LCG (line 7) and decomposes the index into an assignment for each variable (line 10). We calculate the joint probability of p($\mathbf{X}$) for this assignment (line 13) add this probability to the corresponding result row, and increment the sample count.

At any point, the algorithm may be interrupted and each individual probability in $p(\mathbf{X}_q)$ may be estimated from the accumulated probability mass $p(x)$:

$$p(\mathbf{X}_q) = \frac{\prod_{X_j \in \mathbf{X}} |dom(X_j)|}{count_x} \cdot p(x)$$

Cyclic sampling promises to be a good foundation for CIA, but must satisfy the two constraints. First, it needs to provide an epsilon-delta approximation in a fixed period of time. In classical OLA and some approximate inference algorithms, samples generated are independently and identically distributed. As a result, Hoeffding's inequality can provide accuracy guarantees. In CIAs, samples are generated without replacement. We need to provide an $\epsilon - \delta$ approximation under this assumption. Second, cyclic sampling needs to eventually converge to an exact inference result. This requires that we sample all the items in the joint probability distribution exactly once and the samples should be sampled randomly.

Algorithm 2 `CyclicSampling`($\mathcal{B}$, Q)

```
Require: A bayes net $\mathcal{B}$=($\mathcal{G}(\mathbf{X})$,$\Phi$)
Require: A conditional probability query: Q=$P(\mathbf{X_q})$
Ensure: Probabilities for each $\vec{q}$: $\{p_{\vec{q}} = P(\mathbf{X_q}) = \vec{x}_q)\}$
Ensure: The number of samples for each $\vec{q}$: $\{count_{\vec{q}}\}$
 1: $totalSamples \leftarrow 1$
 2: for each $\phi_i$ in $\Phi$ do
 3:     $totalSamples = totalSamples * |dom(X_i)|$
 4: $index_1 \leftarrow$ rand_int() mod $totalSamples$
 5: $a, b, m \leftarrow$ InitLCG($totalSamples$)
 6: for each $k \in 0 \ldots totalSamples$ do
 7:     $assignment \leftarrow index_k$
 8:     /* De-multiplex the variable assignment */
 9:     for each $j \in 0 \ldots n$ do
10:         $x_j \leftarrow assignment \mod |dom(X_j)|$
11:         $assignment \leftarrow assignment \div |dom(X_j)|$
12:     /* probability is a product of the factors */
13:     $prob \leftarrow \prod_i \phi_i \left(\pi_{scope(\phi_i)}(\langle x_1, \ldots, x_n \rangle)\right)$
14:     /* assemble return values */
15:     $\vec{q} \leftarrow \pi_{\mathbf{X_q}}(\ \vec{x}\ )$; $p_{\vec{q}} \leftarrow p_{\vec{q}} + prob$; $count_{\vec{q}} \leftarrow count_{\vec{q}} + 1$
16:     /* step the LCG */
17:     $index_{k+1} \leftarrow (a * index_k + b) \mod m$
```

Computation Cost. Let n be the number of random variables in $\mathcal{B}$, as a simplification assume w.l.o.g. that each random variable has domain size *dom*. Calculating the parameters for LCG takes constant time. The sampling process takes $O(|N|)$ time, where $|N|$ is the total number of samples in the reduced joint probability distribution $P(\mathbf{X})$. N can be as large as dom^n, that is exponential in the size of the graph.

Confidence Bound. Classical approximate inference and OLA algorithms use random sampling with replacement, making it possible to use well known accuracy bounds. For example one such bound, based on Hoeffding's inequality [19] establishes a tradeoff between the number of samples needed n, a probabilistic upper bound on the absolute error ϵ, and an upper bound on probably that the bound will be violated δ. Given two values, we can obtain the third.

Hoeffding's inequality for processes that sample with replacement was extended by Serfling et al. [37] for sampling without replacement. Denote by N the total number of samples, P(x) is the true probability distribution after seeing all the samples N. Denote by $P_n(x)$ the approximation of P(x) after n samples. From [37], and given that probabilities are in general bounded as $0 \le p \le 1$, we have that:

$$P_n\left(P_n(x) \notin [P(x)-\epsilon, P(x)+\epsilon]\right) \le \delta \equiv exp\left[\frac{-2n\epsilon^2}{1-(\frac{n-1}{N-1})}\right] \tag{2}$$

In other words, after n samples, there is a $(1-\delta)$ chance that our estimate $P_n(x)$ is within an error bound ϵ.

3.2 Leaky Joins

In Cyclic Sampling, samples are drawn from an extremely large joint relation and require exponential time for convergence. To address this limitation, we first return to Variable Elimination as described in Section 2.2.1. Recall the clique tree representation in Figure 2 for the BN in Figure 1, where the marginal for the goal variable (J) is produced by clique cluster C_3. Each clique focuses on a single clique variable X_i, and the clique cluster is a product of the separator table to the clique's left and all remaining factor tables containing X_i. As a result, each factor ϕ in $\mathcal{B}$=($\mathcal{G}(\mathbf{X})$,Φ) belongs to exactly one clique cluster. Variable Elimination (the process below the red line) mirrors classical blocking aggregate-join evaluation, computing each separator table (aggregate) fully and passing it to the right.

The key idea is to create a clique tree as in Variable Elimination, but to allow samples to gradually "leak" through the clique tree rather than computing each separator table as a blocking operation. To accomplish this, we propose a new *Leaky Join* relational operator. A single Leaky Join computes a group-by aggregate over one or more Natural Joins, "online" using cyclic sampling as described above. As the operator is given more cpu-time, its estimate improves. Crucially, Leaky Joins are composable. During evaluation, all Leaky Joins in a query plan are updated in parallel. Thus the quality of a Leaky Join operator's estimate is based not only on how many samples it has produced, but also on the quality of the estimates of its input tables.

Algorithm 3 gives an evaluation strategy for inference queries using Leaky Joins. Abstractly, an evaluation plan consists of a set of intermediate tables for each clique $C_i \in \mathbf{C}$ (i.e., each intermediate join), and for each separator $S_i \in \mathbf{S}$ (i.e., each intermediate aggregate). Queries are evaluated volcano-style, iterating over the rows of each clique and summing over the product of probabilities as described in Section 2.2. As in Cyclic Sampling, the iteration order is randomized by a LCG (lines 9-13). For each clique C_i, the algorithm samples a row $\vec{x}$ (lines 9-12), computes the marginal probability for that row (line 14), and adds it to its running aggregate for the group $\vec{q}$ that $\vec{x}$ belongs to (line 20). It is necessary to avoid double-counting samples in the second and subsequent cycles of the LCG. Consequently, the algorithm updates the separator using the difference δ_{prob} between the newly computed marginal and the previous version (lines 16-19).

In order to determine progress towards convergence, the algorithm also tracks a sample count for each row of the clique and separator tables (lines 15, 17), as well as the total, aggregate count for each separator table (line 21). Informally, this count is the number of distinct tuple lineages represented in the current estimate of the probability value. For a given clique table C_i the maximum value of this count is the product of the sizes of the domains of all variables eliminated (aggregated away) in C_i (line 3). For example, in Figure 2, no variables have been eliminated in C_1 so each row of C_1 contains at most one sample. By C_2, the variable D has been eliminated, so each row of C_2 can represent up to $|dom(D)| = 2$ samples. Similarly each row of C_3 represents up to $|dom(D)| \cdot |dom(I)| = 4$ samples. The algorithm uses the sample count as a termination condition. Once a table is fully populated, sampling on it stops (line 8). Once all tables are fully populated, the algorithm has converged (line 7).

Algorithm 3 EvaluateLeakyJoins($\mathcal{B}$, Q)

```
**Require:** A bayes net $\mathcal{B} = (\mathcal{G}(\mathbf{X}), \Phi)$
**Require:** An inference query $Q = P(\mathbf{X_q})$
**Ensure:** The result separator $S_{target} = P(\mathbf{X_q})$
 1: $\langle \mathbf{S}, \mathbf{C} \rangle \leftarrow$ assemblePlan($\mathcal{B}, \mathbf{X}_q$)
 2: for each $i \in 1 \ldots |\mathbf{S}|$ do
 3:     $samples_i \leftarrow 0$;   $maxSamples_i = |dom(desc(S_i))|$
 4:     $a_i, b_i, m_i \leftarrow$ initLCG($|dom(C_i)|$)
 5:     $index_i \leftarrow$ rand_int() mod $m_i$
 6:     Fill $S_i$ and $C_i$ with $\langle$ $prob : 0.0$ , $count : 0$ $\rangle$
 7: while there is an $i$ with $samples_i < maxSamples_i$ do
 8:     for each $i$ where $samples_i < maxSamples_i$ do
 9:         /* Step the LCG */
10:         $index_i \leftarrow (a_i * index_i + b_i)$ mod $|dom(\psi_i)|$
11:         /* Demux index as Alg. 2 lines 9-11 */
12:         Get $\vec{x}$ from $index_i$
13:         /* Get the joint probability as Alg. 2 line 13
                and get the joint sample count similarly */
14:         $prob \leftarrow \prod_{\phi \in factors(C_i)} \left(\phi\left[\pi_{scope(\phi)}(\vec{X})\right].prob\right)$
15:         $count \leftarrow \prod_{\phi \in factors(C_i)} \left(\phi\left[\pi_{scope(\phi)}(\vec{X})\right].count\right)$
16:         /* Compute update deltas */
17:         $\langle \delta_{prob}, \delta_{count} \rangle = \langle$ $prob, count$ $\rangle - C_i[\vec{x}]$
18:         /* Apply update deltas */
19:         $C_i[\vec{x}] = C_i[\vec{x}] + \langle \delta_{prob}, \delta_{count} \rangle$
20:         $\vec{q} = \pi_{scope(S_i)}(\vec{x})$;      $S_i[\vec{q}] = S_i[\vec{q}] + \langle \delta_{prob}, \delta_{count} \rangle$
21:         $samples_i = samples_i + \delta_{count}$
```

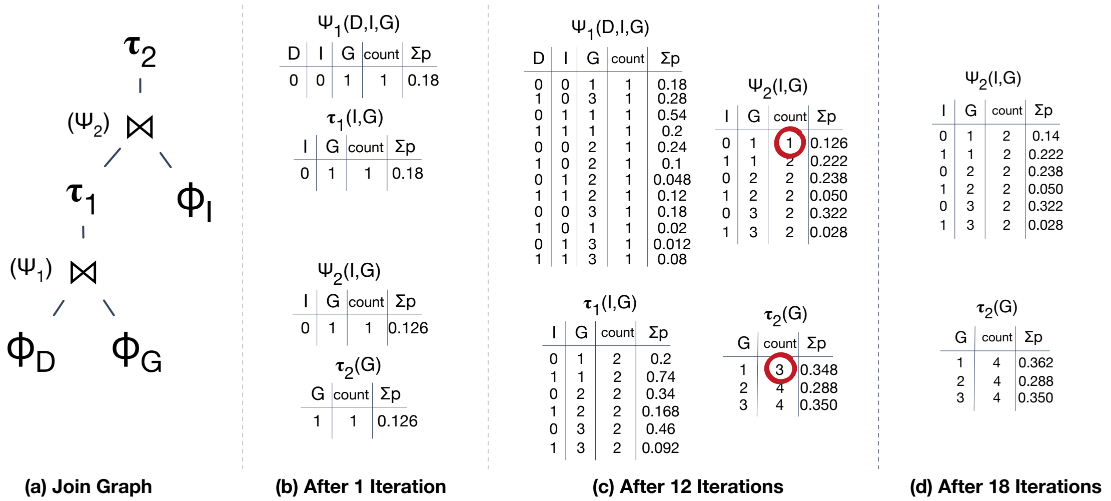
In short, Leaky Joins work by trickling samples down through each level of the join graph. The cyclic sampler provides flow control and acts as a source of randomization, allowing all stages to produce progressively better estimates in parallel. With each cycle through the samples, improved estimates from the join operator's input are propagated to the next tier of the query.

Example 3. *As an example of Leaky Joins, consider a subset of the graph in Figure 1 with only nodes D, I, G and an inference query for $p(G)$. Using classical heuristics from variable elimination, the Leaky Joins algorithm elects to eliminate D first and then I, and as a result assembles the intermediate clique cluster $\boldsymbol{C}$ and clique separator $\boldsymbol{S}$ tables as shown in Figure 3. Samples are generated for each intermediate clique cluster one at a time, following the join order: First from $C_1(D, I, G)$ and then $C_2(I, G)$. As shown in Figure 3b, the first sample we obtain from C_1 is $\langle 0, 0, 1 \rangle$*

$$\phi_D(D=0) \cdot \phi_G(D=0, I=0, G=1) = 0.6 \cdot 0.3 = 0.18$$

$S_1(I, G)$ is correspondingly updated with the tuple $\langle 0, 1 \rangle$ with aggregates $\langle 1, 0.18 \rangle$. Then the second sample is drawn from $C_2(I, G)$. For this example, we will assume the random sampler selects $\langle 0, 1 \rangle$, which has probability:

$$S_1(I=0, G=1) \cdot \phi_I(I=0)$$



(a) Join Graph

Ψ1(D,I,G)

D	I	G	count	Σp
0	0	1	1	0.18

τ1(I,G)

I	G	count	Σp
0	1	1	0.18

Ψ2(I,G)

I	G	count	Σp
0	1	1	0.126

τ2(G)

G	count	Σp
1	1	0.126

(b) After 1 Iteration

Ψ1(D,I,G)

D	I	G	count	Σp
0	0	1	1	0.18
1	0	3	1	0.28
0	1	1	1	0.54
1	1	1	1	0.2
0	0	2	1	0.24
1	0	2	1	0.1
0	1	2	1	0.048
1	1	2	1	0.12
0	0	3	1	0.18
1	0	1	1	0.02
0	1	3	1	0.012
1	1	3	1	0.08

Ψ2(I,G)

I	G	count	Σp
0	1	1	0.126
1	1	2	0.222
0	2	2	0.238
1	2	2	0.050
0	3	2	0.322
1	3	2	0.028

τ1(I,G)

I	G	count	Σp
0	1	2	0.2
1	1	2	0.74
0	2	2	0.34
1	2	2	0.168
0	3	2	0.46
1	3	2	0.092

τ2(G)

G	count	Σp
1	3	0.348
2	4	0.288
3	4	0.350

(c) After 12 Iterations

Ψ2(I,G)

I	G	count	Σp
0	1	2	0.14
1	1	2	0.222
0	2	2	0.238
1	2	2	0.050
0	3	2	0.322
1	3	2	0.028

τ2(G)

G	count	Σp
1	4	0.362
2	4	0.288
3	4	0.350

(d) After 18 Iterations

Figure 3: Leaky Joins example join graph (a) and the algorithm's state after 1, 12, and 18 iterations (b-d). In the 12-iteration column (c), incomplete sample counts are circled.

Although we do not have a precise value for S_1 we can still approximate it at this time and update S_2 accordingly. After 12 samples, C_1 has completed a round of sampling and is ready to be finalized. The state at this point is shown in Figure 3c. Note that the approximation of S_2 is still incorrect — The approximation made in step 1 and several following steps resulted in only partial data for $\psi_2(I = 1, G = 1)$ (circled counts in Fig. 3c). However, this error will only persist until the next sample is drawn for $C_2(0, 1)$, at which point the system will have converged to a final result.

Cost Model. We next evaluate the cost of reaching an exact solution using the Leaky Join algorithm. Assume we have k random variables $X_1, ..., X_k$, and the corresponding k factors $\phi_{X_1}, ..., \phi_{X_k}$. Furthermore, assume the variables are already arranged in the optimal elimination order, and we wish to marginalize out variables $X_1, \ldots, X_j$. Leaky joins generates exactly the same set of j cliques and separators as Variable Elimination. Like variable elimination, we can measure the computation complexity by counting multiplication and addition steps. The primary difference between Variable Elimination and Leaky Joins is that some aggregation steps will base on approximations and must be repeated multiple times. Let V denote the cost of constructing the largest joint factor in Variable Elimination (i.e., the time complexity of Variable Elimination is $O(V)$). After V iterations, the lowest level of the join tree is guaranteed to be finalized. After a successive V iterations, the second level of the tree is guaranteed to be finalized, and so forth. The maximum depth of the join tree is the number of variables k, so a loose bound for the complexity Leaky Joins is $O(kV)$.

Confidence Bound. In Section 3.1, we showed an ϵ-δ bound for random sampling without replacement (Formula (2)). Here, we extend this result to give a loose bound for Leaky Joins. The primary challenge is that, in addition to sampling errors in the Leaky Join itself, the output can be affected by cumulative error from the join's inputs. We consider the problem recursively. The base case is a clique that reads from only input factors — the lowest level of joins in the query plan. Precise values for inputs are available immediately and Formula (2) can be used as-is.

Next, consider a clique C_2 computed from only a single leaky join output. Thus, we can say that $C_2 = S_1 \bowtie \phi$, where ϕ is the natural join of all input factors used by C_2. There are $|dom(S_1)|$ rows in S_1, so after n sampling steps, each row of S_1 will have received $\frac{n}{|dom(S_1)|}$ samples. Denote the maximum number of samples per row of S_1 by $N_1 = \frac{|dom(S_1)|}{|dom(C_1)|}$. Then, by (2), all rows in S_1 will have error less than ϵ with probability:

$$\delta_1 \equiv \delta^{|dom(S_1)|} = exp\left[\frac{-2n\epsilon^2 \cdot (N_1 - 1)}{N_1 - \frac{n}{|dom(S_1)|}}\right] \qquad (3)$$

Let us consider a trivial example where the cumulative error in each row of S_1 is bounded by ϵ:

S_1	X_1	p
	1	$p_1 \pm \epsilon$
	2	$p_2 \pm \epsilon$

ϕ	X_1	p
	1	p_3
	2	p_4

Here, the correct joint probabilities for rows of C_2 are $p_1 \cdot p_3$ and $p_2 \cdot p_4$ respectively. Thus a fully-sampled S_2 (projecting away X_1) will be approximated as $(p_1 \pm \epsilon)p_3 + (p_2 \pm \epsilon)p_4$. The cumulative error in this result is $(p_3 + p_4)\epsilon$, or using a pessimistic upper bound of 1 for each p_i, at worst 2ϵ. Generalizing, if one row of S_2 is computed from k rows of C_1, the cumulative error in a given row of S_2 is at most $k\epsilon$. Repeating (3), sampling error on S_2 after n rounds will be bounded by ϵ with probability $\delta^{|dom(S_2)|}$. After n rounds of sampling, each row of S_2 will have received $\frac{n}{|dom(S_2)|}$ rows of C_2, so the cumulative error on one row is $\frac{n}{|dom(S_2)|}\epsilon$. Combining (2) and (3), we get that for one row of S_2:

$$P\left[|p - \mathbb{E}_{n,x}| < \left(1 + \frac{n}{|dom(S_2)|}\right)\epsilon\right] \leq$$

$$exp\left[\frac{-2\frac{n}{|dom(S_2)|}\epsilon^2 \cdot (N_2 - 1)}{N_2 - \frac{n}{|dom(S_2)|}}\right] \cdot \delta_1 \qquad (4)$$

The joint probability across all rows of S_2 is thus:

$$exp\left[\frac{-2n\epsilon^2(N_2 - 1)}{N_2 - \frac{n}{|dom(S_2)|}}\right] \cdot \delta_1^{|dom(S_2)|} \equiv \delta_2 \cdot \delta_1^{|dom(S_2)|}$$

Generalizing to any **left-deep** plan, an error ϵ' defined as:

$$\epsilon' = \epsilon \cdot \prod_{i=2}^{|\mathbf{X}|} \left(1 + \frac{n}{|dom(S_i)|}\right) \tag{5}$$

is an upper bound on the marginal in $S_{|\mathbf{X}|}$ with probability:

$$\prod_{i=1}^{|\mathbf{X}|} \delta_i^{\left(\prod_{j=1}^{i-1} |dom(S_j)|\right)} = \prod_{i=1}^{|\mathbf{X}|} e^{\left(\left(\prod_{j=1}^{i-1} |dom(S_j)|\right) \frac{-2n\epsilon^2 (N_j - 1)}{N_j - \frac{n}{|dom(S_j)|}}\right)} \tag{6}$$

Consider a slightly more complicated toy example clique $C_4 = S_3 \bowtie S_1$, where both S_3 and S_1 both have bounded error ϵ_1 and ϵ_3 respectively.

C_1	X_1	p
	1	$p_1 \pm \epsilon_1$
	2	$p_2 \pm \epsilon_1$

C_3	X_1	p
	1	$p_3 \pm \epsilon_3$
	2	$p_4 \pm \epsilon_3$

As before, the correct joint probability is $p_1 \cdot p_3 + p_2 \cdot p_4$. Given an $\epsilon' = \max(\epsilon_1, \epsilon_3)$, the estimated probability will be $p_1 \cdot p_3 + p_2 \cdot p_4 + \left(\sum_{i=1}^{4} p_i \epsilon'\right) + \epsilon'^2$. As before, using an upper bound of 1 for each $p_1 \ldots p_4$ bounds the error by:

$$(|dom(S_1)| \cdot |dom(S_3)|)\epsilon' + \epsilon'^2$$

More generally, the predicted value across m source tables is a sum of terms of the form $(p+\epsilon')$, and the overall cumulative error per element of C_1 is bounded as:

$$\epsilon_{cum} = \sum_{i=1}^{m-1} (m\ \mathcal{C}\ i)\epsilon^{m-i}$$

where $m\ \mathcal{C}\ i$ is the combinatorial operator m choose i. Thus re-using Equation (4), we can solve the recursive case of a separator with m leaky join inputs that each have error bounded by ϵ' with probability $\delta_{cum} = \prod_i^m \delta_i$. Then the total error on one row of the separator can be bounded by $\epsilon + \epsilon_{cum}$ with probability:

$$exp\left[\frac{-2\frac{n}{|dom(S_2)|}\epsilon^2 \cdot (N_2 - 1)}{N_2 - \frac{n}{|dom(S_2)|}}\right] \cdot \delta \tag{7}$$

The joint error is computed exactly as before. To estimate the error on the final result, we apply this formula recursively on the full join plan.

4. LESSONS LEARNED FROM IVM

Materialized views are the precomputed results of a so-called view query. As the inputs to this query are updated, the materialized results are updated in kind. Incremental view maintenance (IVM) techniques identify opportunities to compute these updates more efficiently than re-evaluating the query from scratch. Incremental view maintenance has already seen some use in Monte Carlo Markov Chain inference [43], and recent advances — so called recursive IVM techniques [4, 25] have made it even more efficient.

Our initial attempts at convergent inference were based on IVM and recursive IVM in particular. It eventually became clear that there was a fundamental disconnect between these techniques and the particular needs of graphical inference. In the interest of helping others to avoid these pitfalls, we use this section to outline our basic approach and to explain why, perhaps counter-intuitively, both classical and recursive IVM techniques are a poor fit for convergent inference on graphical models.

4.1 The Algorithm

Our first approach at convergent inference used IVM to compute and iteratively revise an inference query over a progressively larger fraction of the input dataset. That is, we declared the inference query as a materialized view using exactly the query defined in Section 2.2. The set of factor tables was initially empty. As in Cyclic Sampling, we iteratively insert rows of the input factor tables in a shuffled order. A backend IVM system updates the inference result, eventually converging to a correct value once all factor rows have been inserted. This process is summarized in Algorithm 4.

Algorithm 4 `SimpleIVM-CIA`($\mathcal{B}$, Q)

```
Require: A bayes net $\mathcal{B}$=($\mathcal{G}(\mathbf{X})$,P)
Require: A conditional probability query: Q=$P(\mathbf{X_q})$
Ensure: The set $ret_{\mathbf{X_q}} = P(\mathbf{X_q})$
for each $\phi_i \in G(\mathbf{X})$ do
    $index_{0,i}$ = rand_int() mod $|dom(X_i)|$
    $a_i, b_i, m_i \leftarrow$ InitLCG($|dom(X_i)|$)
    $\phi'_i \leftarrow \emptyset$
Compile IVM Program $Q'$ to compute $P(\mathbf{X_q})$ from $\{\phi'_i\}$
for each $k \in 1 \ldots max_i(|dom(X_i)|)$ do
    for each $\phi_i \in G(\mathbf{X})$ do
        if $k \leq |dom(X_i)|$ then
            $index_{k,i} \leftarrow (a * index_{k-1,i} + b)$ mod $m_i$
            Update $Q'$ with row $\phi'_i[index_{k,i}] \leftarrow \phi_i[index_{k,i}]$
$ret_{\mathbf{X_q}} \leftarrow$ the output of $Q'$
```

While naive cyclic sampling samples directly from the output of the join, IVM-CIA constructs the same output by iteratively combining parts of the factor tables. The resulting update sequence follows a pattern similar to that of multi-dimensional Ripple Joins [17], incrementally filling the full sample space of the join query. As in naive cyclic sampling, this process may be interrupted at any time to obtain an estimate of $P(\mathcal{Y})$ by taking the already materialized partial result and scaling it by the proportion of samples used to construct it. This proportion can be computed by adding a `COUNT(*)` aggregate to each query. IVM-CIA uses the underlying IVM engine to simultaneously track both the estimate and the progress towards an exact result.

4.2 Post-Mortem

For our first attempt at an IVM-based convergent inference algorithm, we used DBToaster [25], a recursive IVM compiler. DBToaster is aimed at relational data and uses a sparse table encoding that, as we have already mentioned, is ill suited for graphical models. Recognizing this as a bottleneck, we decided to create a modified version of DBToaster that used dense array-based table encodings. Although this optimization did provide a significant speed-up, the resulting engine's performance was still inferior: It converged much slower than variable elimination and had a shallower result quality ramp than the approximation techniques (even cyclic sampling in some cases).

Ultimately, we identified two key features of graphical models that made them ill-suited for database-centric IVM and in particular recursive IVM. First, in Section 3, we noted that nodes in a Bayesian Network tend to have many neighbors and that the network tends to have high hypertree-width. The size of intermediate tables is exponential in the hypertree-width of the query — rather large in the case of graphical models. Recursive IVM systems like DBToaster are in-effect a form of dynamic programming, improving performance by consuming more space. DBToaster in particular maintains materialized copies of intermediate tables for all possible join plans. As one might imagine, the space required for even a BN of moderate size can quickly outpace the available memory.

The second, even more limiting factor of IVM for graphical models is the fan-out of joins. Because of the density of a graphical model's input factors and intermediate tables, joins are frequently not just many-many, but all-all. Thus a single insertion (or batch of insertions) is virtually guaranteed to affect every result row. In recursive IVM, the problem is worse, as each insertion can trigger full-table updates for nearly every intermediate table (which as already noted, can be large). Batching did improve performance, but not significantly enough to warrant replacing cyclic sampling.

Our approach of Leaky Joins was inspired, in large part, by an alternative form of recursive IVM initially described by Ross et. al., [32], which only materializes intermediates for a single join plan. Like this approach, Leaky Joins materializes only a single join plan, propagating changes through the entire query tree. However, unlike the Ross recursive join algorithm, each Leaky Join operator acts as a sort of batching blocker. New row updates are held at each Leaky Join, and only propagated in a random order dictated by the LCG.

5. EVALUATION

Recall in Section 3, we claimed CIAs should satisfy four properties. In this section we present experimental results to show that they do. Specifically, we want to show: (1,2) **Flexibility:** CIAs are able to provide both approximate results and exact results in the inference process. (3) **Approximation Accuracy:** Given the same amount of time, CIAs can provide approximate results with an accuracy that is competitive with state-of-the-art approximation algorithms. (4) **Exact Inference Efficiency:** The time a CIA takes to generate an exact result is competitive with state-of-the-art exact inference algorithms.

5.1 Experimental Setup and Data

Experiments were run on a 12 core, 2.5 GHz Intel Xeon with 198 GB of RAM running Ubuntu 16.04.1 LTS. Our experimental code was written in Java and compiled and run single-threaded under the Java HotSpot version 1.8 JDK/JVM. For experiments, we used five probabilistic graphical models from publicly available sources, including the bnlearn Machine Learning Repository [35] to compare the available algorithms. Visualizations of all five graphs are shown in Figure 4.

Student. The first data set is the extended version of the Student graphical model from [26]. This graphical model contains 8 random variables. All the random variables are discrete. In order to observe how CIAs are influenced by exponential blowup of scale, we use this graph as a microbenchmark by generating synthetic factor tables for the Student graph. In the synthetic data, we vary the domain size of each random variable from 2 to 25. Marginals were computed over the `Happiness` attribute.

Child. The second graphical model captures the symptoms and diagnosis of Asphyxia in children [12]. The number of random variables in the graph is 20. All the random variables are discrete with different domain sizes. There are 230 parameters in factors. The average degree of nodes in the graph is 3.5 and the maximum in-degree in the graph is 4. Marginals were computed over the `sick` variable.

Insurance. The third graphical model we used models potential clients for car insurance policies [8]. It contains 27 nodes. All the random variables are discrete with different domain sizes. There are 984 parameters in factors. The average number of degree is 3.85 and the maximum in-degree in the graph is 3. Marginals were computed over the `PropCost` variable.

Barley. The fourth graphical model is developed from a decision support system for mechanical weed control in malting barley [27]. The graph contains 48 nodes. All the random variables are discrete with different domain sizes. There are 114005 parameters in factors. The average degree of nodes in the graph is 3.5 and the maximum in-degree in the graph is 4. Marginals were computed over the `ntilg` variable.

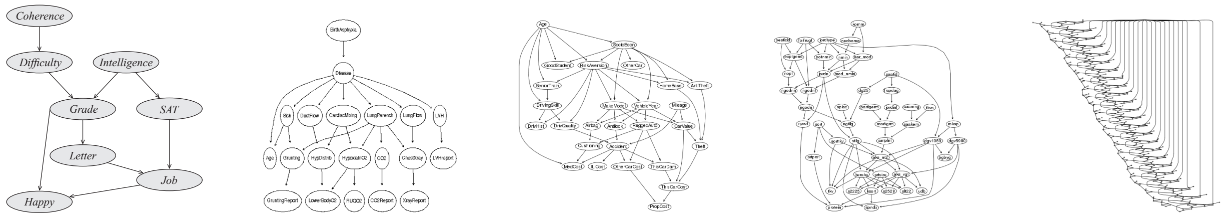
Diabetes. The fifth graphical model is a very large graph that captures a model for adjusting insulin [5]. The number of random variables in the graph is 413. All the random variables are discrete with different domain sizes. There are 429409 parameters in factors. In average, there are 2.92 degrees and the maximum in-degree in the graph is 2. Marginals were computed over the `bg_5` variable.

5.2 Inference Methods

We compare our two convergent inference algorithms with variable elimination (for exact inference results) and Gibbs sampling (for approximate inference). We are interested in measuring runtime for exact inference and accuracy for approximate inference. We assign an index for each node in the Bayes net following the topological order. We assume that for each factor ϕ_i, the variables are ordered by the following conventions: $pa(\mathbf{X}) \prec \mathbf{X}$, where $\mathbf{X}=\{X_i, \ldots, X_j, \ldots\}$ is ordered increasingly by index. The domain values in each random variable is ordered increasingly.

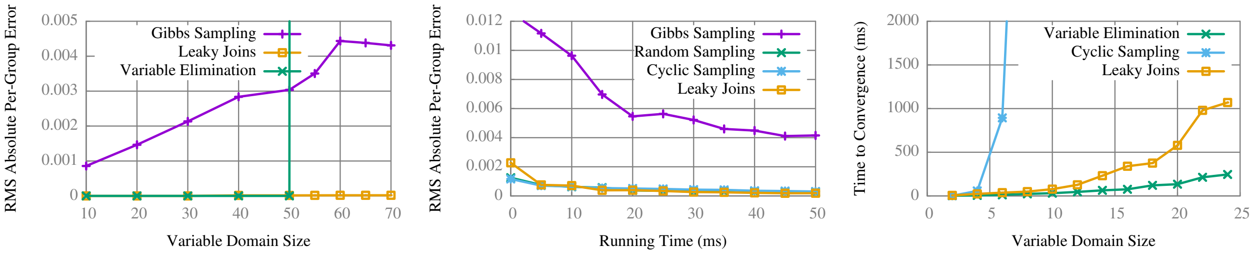
Variable Elimination. Variable elimination is the classic exact inference algorithm used for graphical models, as detailed in Section 2. As is standard in variable elimination, intermediate join results are streamed and never actually materialized. To decide on an elimination (join) ordering, we adopt the heuristic methods in [26]. At each point, the algorithm evaluates each of the remaining variables in the Bayes net based on a heuristic cost function. The cost criteria used for evaluating each variable is Min-weight, where the cost of a node is the product of weights, where weight is the domain cardinality of the node's neighbors. For example, using Min-weight, the selected order for the extended student graph in Figure 4a is: C, D, S, I, L, J, G. H is the target random variable.

Gibbs Sampling. As discussed in Section 2, Gibbs sampling first generates the initial sample with each variable X_i in $\mathcal{B} = (\mathcal{G}(\mathbf{X}), \Phi)$ following the conditional probability distribution $p(X_i|pa(X_i))$. Then, we randomly fix the value for some random variable X_j and use it as evidence to generate next sample. With more and more samples collect, the distribution will get increasingly closer to the posterior. We skip the first hundred samples for small graphs 4a, 4b and 4c, and thousand samples for larger graphs 4d and 4e at the beginning of the sample process. The target probability



(a) Extended Student (b) Child (c) Insurance (d) Barley (e) Diabetes

Figure 4: Visualizations of five graphical models from [35] used in our experiments.



(a) Accuracy within 10ms (b) Accuracy over time (c) Running time to convergence

Figure 5: Microbenchmarks on the synthetic, extended Student graph (Figure 4a)

distribution is calculated by normalizing the sum of sample frequencies for each value in the target variables X_q.

Cyclic Sampling. Cyclic Sampling is our first CIA, described in Section 3.1 and in Algorithm 2. Note that cyclic sampling does not materialize the joint probability distribution, but rather constructs rows of the joint distribution dynamically using a LCG (Equation 1). This process takes constant time for a fixed graph.

Leaky Joins. Leaky Joins, our second CIA, were described in Section 3.2. We use the same elimination (join) order as in variable elimination algorithm to construct the clique tree and materialize intermediate tables, Clique's clusters **C** and Clique's seperator **S**. Then we conduct the "passing partial messages" process according to Algorithm 3.

5.3 Flexibility

We first explore the flexibility of CIAs by comparing the accuracy of different algorithms (both exact and approximate) within a finite cutoff time. We imitate the situation that time is the major concern for user and the goal is to provide an accurate inference result at a given time. We compute the marginal probability, cutting each algorithm off after the predefined period, and average the fractional error across each marginal "group".

Figure 5a shows the average fractional error for each inference algorithm on the **student** graph with a cutoff of 10 seconds. We vary the factor size from 10 to 70 to simulate small and large graphs. Variable elimination provides an exact inference result for variables with domain size smaller than 50. On larger graphs, it times out, resulting in a 100% error. Gibbs sampling can always provide an approximate result, but produces results that are inaccurate, even when variable elimination can produce an exact result. Leaky joins provide exact inference results when the domain size is smaller than 40, but when the domain size passes 40, it still provides approximate results with a lower error than Gibbs sampling. This graph shows that, with leaky joins, the same algorithm can support both the exact inference and approximate inference cases; neither users or inference engines need to anticipate which class of algorithms to use.

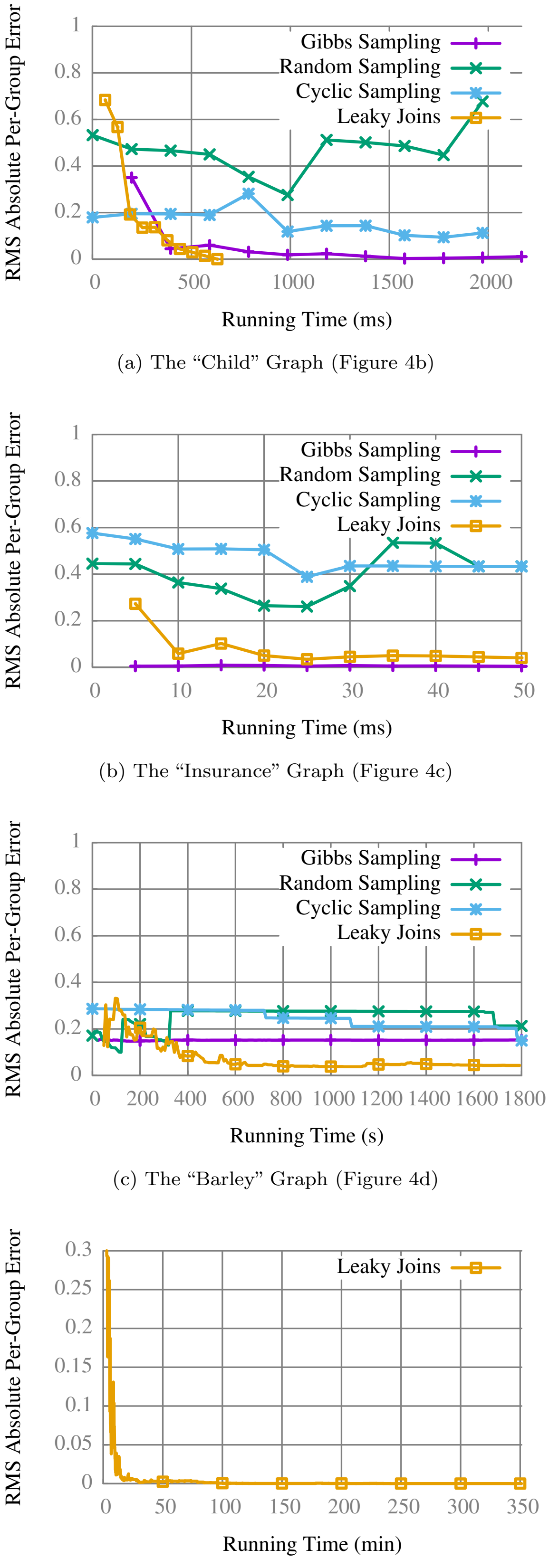
5.4 Approximate Inference Accuracy

Figure 5b compares each algorithm's approximate accuracy relative to time spent on the **student** graph with a node size of 35. At each time t, the average fractional error of leaky joins is smaller than that of Gibbs sampling. In addition, leaky join converges to exact result, which Gibbs sampling will never do. The steep initial error in leaky joins at the start stems from the first few sampling rounds for each intermediate table C_i being based on weak preliminary approximations; The algorithm needs some burn-in time to have samples to cover their domains to provide approximate results. Gibbs sampling algorithm also has burn-in process. The sharp curve for Gibbs sampling is because it takes more samples for Gibbs sampling to cover corner cases, and generate samples with less probabilities. Figure 6a, Figure 6b and Figure 6d show the approximate accuracy result for child and insurance graph. Gibbs sampling performs well in this two graph for that the domain of the target variables X_q are small (the domain of sick node for child graph is two and propcost for insurance is 4). There will be less corner cases and by *Chernoff's bound* [11], the number of samples to required decreases as the probability of $\mathrm{P}(X_q)$ increases.

$$P_{est}(P_{est}(X_q) \notin P(X_q)(1 \pm \epsilon)) \leq 2e^{-NP(X_q)\epsilon^2/3} \leq \delta.$$

The result shows that even in this situation, the approximate result of leaky joins is still comparable to Gibbs sampling. Of these four graphs, the "Diabetes" graph was the most complex, and only Leaky Joins produced meaningful results.

For comparison with the accuracy results in Figure 6, Variable Elimination produces exact results for "Child" in 19 ms, for "Insurance" in 49 ms, for Barley in approximately 1.4 hours, and for Diabetes in approximately 1.5 hours.



(a) The “Child” Graph (Figure 4b)

(b) The “Insurance” Graph (Figure 4c)

(c) The “Barley” Graph (Figure 4d)

(d) The “Diabetes” Graph (Figure 4e)

Figure 6: Approximation accuracy for real-world graphs

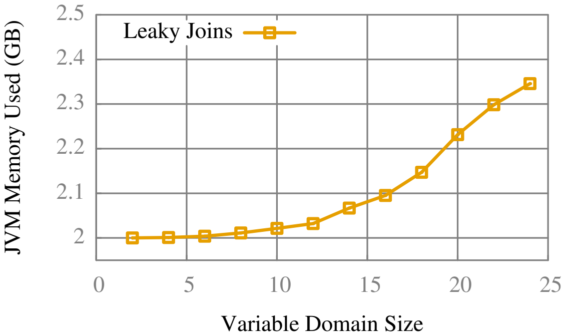


Figure 7: JVM Memory use for Cyclic Sampling. Note that at startup, Java has already allocated roughly 2 GB.

5.5 Convergence Time

Figure 5c shows the exact running time for variable elimination and leaky joins. Recall that the running complexity of variable elimination and leaky join is dominated by the size of the clique's cluster, $O(k|C_{max}|)$, where $|C_{max}|$ is the size of largest clique's cluster. The difference is that leaky join has a constant k. As the factor size increases, C_{max} increases and both algorithms get slower at equivalent rates.

5.6 Memory

Unlike variable elimination and many of the approximate algorithms, leaky joins does continuously maintain materialized intermediate results. To measure memory use, we used the JVM's `Runtime.totalMemory()` method, sampling immediately after results were produced, but before garbage collection could be run. Figure 7 shows Java's memory usage with leaky joins for the “Student” micro benchmark. The actual memory needs of leaky joins are comparatively small: The two largest intermediate results have roughly 20-thousand rows, with 5 columns each. Overall, Leaky Joins only forms a small portion of Java's overall footprint.

6. CONCLUSIONS AND FUTURE WORK

We introduced a class of convergent inference algorithms (CIAs) based on sampling without replacement using linear congruential generators. We proposed CIAs built over incremental view maintenance and a novel aggregate join algorithm that we call Leaky Joins. We evaluated both IVM-CIA and Leaky Joins, and found that Leaky Joins were able to approximate the performance of Variable Elimination on simple graphs, and the accuracy of state-of-the-art approximation techniques on complex graphs. As graph complexity increased, the bounded-time accuracy of Leaky Joins degraded gracefully.

Our algorithms has one limitation which represents opportunities for future work. Our algorithm didn't consider scalability. In the era of Big Data, distributed inference in graphical model is necessary for performance. Similar to works for parallelizing existing inference algorithms [7, 14], our algorithm is possible to run in parallel.

Acknowledgements. This work was supported by a gift from Oracle Academic Relations, by NPS Grant N00244-16-1-0022 and by NSF Grants #1409551 and #1640864. Opinions, findings and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of Oracle, the Naval Postgraduate School, or the National Science Foundation.

7. REFERENCES

[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The Aqua approximate query answering system. *SIGMOD Rec.*, 28(2):574–576, 1999.

[2] S. Agarwal, A. P. Iyer, A. Panda, S. Madden, B. Mozafari, and I. Stoica. Blink and it's done: Interactive queries on very large data. *pVLDB*, 5(12):1902–1905, 2012.

[3] S. Agarwal, A. Panda, B. Mozafari, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. Technical report, ArXiV, 03 2012.

[4] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *pVLDB*, 5(10):968–979, 2012.

[5] S. Andreassen, R. Hovorka, J. Benn, K. G. Olesen, and E. R. Carson. A model-based approach to insulin adjustment. In *AIME 91*, pages 239–248. Springer, 1991.

[6] S. Arumugam, F. Xu, R. Jampani, C. Jermaine, L. L. Perez, and P. J. Haas. MCDB-R: Risk analysis in the database. *pVLDB*, 3(1-2):782–793, 2010.

[7] R. Bekkerman, M. Bilenko, and J. Langford. *Scaling up machine learning: Parallel and distributed approaches.* Cambridge University Press, 2011.

[8] J. Binder, D. Koller, S. Russell, and K. Kanazawa. Adaptive probabilistic networks with hidden variables. *Machine Learning*, 29(2-3):213–244, 1997.

[9] H. C. Bravo and et al. Optimizing mpf queries: Decision support and probabilistic inference, 2007.

[10] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *VLDB*, 1994.

[11] H. Chernoff. A career in statistics. *Past, Present, and Future of Statistical Science*, page 29, 2014.

[12] A. P. Dawid. Prequential analysis, stochastic complexity and bayesian inference. *Bayesian statistics*, 4:109–125, 1992.

[13] A. Deshpande and S. Madden. MauveDB: Supporting model-based user views in database systems. In *SIGMOD*, 2006.

[14] F. Diez and J. Mira. Distributed inference in bayesian networks. *Cybernetics and Systems: An International Journal*, 25(1):39–61, 1994.

[15] A. Dobra, C. Jermaine, F. Rusu, and F. Xu. Turbo-charging estimate convergence in DBO. *pVLDB*, 2(1):419–430, Aug. 2009.

[16] P. Haas. Large-sample and deterministic confidence intervals for online aggregation. In *SSDBM*, pages 51–62, 1997.

[17] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, 1999.

[18] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. *SIGMOD Rec.*, 26(2):171–182, 1997.

[19] W. Hoeffding. Probability inequalities for sums of bounded random variables. *JASS*, 58(301):13–30, 1963.

[20] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. MCDB: A monte carlo approach to managing uncertain data. In *SIGMOD*, 2008.

[21] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the DBO engine. *TODS*, 33(4):23:1–23:54, 2008.

[22] O. Kennedy and S. Nath. Jigsaw: Efficient optimization over uncertain enterprise data. In *SIGMOD*, 2011.

[23] A. Klein, R. Gemulla, P. Rösch, and W. Lehner. Derby/S: A DBMS for sample-based query answering. In *SIGMOD*, 2006.

[24] D. Knuth. The art of computer programming. semi-numerical algorithms. 1968.

[25] 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. *VLDBJ*, 23(2):253–278, 2014.

[26] D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques.* MIT press, 2009.

[27] K. Kristensen and I. Rasmussen. A decision support system for mechanical weed control in malting barley. In *ECITA*, 1997.

[28] P. L'Ecuyer. Random numbers for simulation. *CACM*, 33(10):85–97, 1990.

[29] C. Mayfield, J. Neville, and S. Prabhakar. Eracer: A database approach for statistical inference and data cleaning. In *SIGMOD*, 2010.

[30] F. Olken and D. Rotem. Simple random sampling from relational databases. In *VLDB*, 1986.

[31] S. K. Park and K. W. Miller. Random number generators: good ones are hard to find. *CACM*, 31(10):1192–1201, 1988.

[32] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. *SIGMOD Rec.*, 25(2):447–458, 1996.

[33] F. Rusu and A. Dobra. Glade: A scalable framework for efficient analytics. *OSR*, 46(1):12–18, Feb. 2012.

[34] B. Schneier. *Applied cryptography: protocols, algorithms, and source code in C.* Wiley & Sons, 2007.

[35] M. Scutari. The bnlearn bayesian network repository. http://www.bnlearn.com/bnrepository/.

[36] P. Sen, A. Deshpande, and L. Getoor. Prdb: Managing and exploiting rich correlations in probabilistic databases. *VLDB Journal, special issue on uncertain and probabilistic databases*, 2009.

[37] R. J. Serfling. Probability inequalities for the sum in sampling without replacement. *The Annals of Statistics*, pages 39–48, 1974.

[38] F. L. Severence. *System modeling and simulation: an introduction.* John Wiley & Sons, 2009.

[39] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. *The Architecture of SciDB.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[40] D. Z. Wang, Y. Chen, C. E. Grant, and K. Li. Efficient in-database analytics with graphical models. *IEEE Data Eng. Bull.*, 37(3):41–51, 2014.

[41] D. Z. Wang, M. J. Franklin, M. Garofalakis, J. M. Hellerstein, and M. L. Wick. Hybrid in-database inference for declarative information extraction. In *SIGMOD*, 2011.

[42] D. Z. Wang, E. Michelakis, M. Garofalakis, and J. M. Hellerstein. Bayesstore: Managing large, uncertain data repositories with probabilistic graphical models. *pVLDB*, 1(1):340–351, 2008.

[43] M. Wick, A. McCallum, and G. Miklau. Scalable probabilistic databases with factor graphs and mcmc. *pVLDB*, 3(1-2):794–804, 2010.

[44] M. L. Wick, A. McCallum, and G. Miklau. Scalable probabilistic databases with factor graphs and MCMC. *CoRR*, abs/1005.1934, 2010.

[45] S. Wu, C. Zhang, F. Wang, and C. Ré. Incremental knowledge base construction using deepdive. *CoRR*, abs/1502.00731, 2015.

Efficient Motif Discovery in Spatial Trajectories Using Discrete Fréchet Distance

Bo Tang* Man Lung Yiu* Kyriakos Mouratidis# Kai Wang$

*The Hong Kong Polytechnic University #Singapore Management University $Zhejiang University

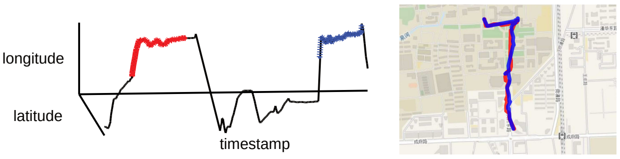
{csbtang,csmlyiu}@comp.polyu.edu.hk; kyriakos@smu.edu.sg; kaelwang@zju.edu.cn

ABSTRACT

The discrete Fréchet distance (DFD) captures perceptual and geographical similarity between discrete trajectories. It has been successfully adopted in a multitude of applications, such as signature and handwriting recognition, computer graphics, as well as geographic applications. Spatial applications, e.g., sports analysis, traffic analysis, etc. require discovering the pair of most similar subtrajectories, be them parts of the same or of different input trajectories. The identified pair of subtrajectories is called a motif. The adoption of DFD as the similarity measure in motif discovery, although semantically ideal, is hindered by the high computational complexity of DFD calculation. In this paper, we propose a suite of novel lower bound functions and a grouping-based solution with multi-level pruning in order to compute motifs with DFD efficiently. Our techniques apply directly to motif discovery within the same or between different trajectories. An extensive empirical study on three real trajectory datasets reveals that our approach is 3 orders of magnitude faster than a baseline solution.

1. INTRODUCTION

Spatial trajectories are prevalent in many applications, e.g., moving object analysis, traffic estimation and prediction systems. In this paper, we study motif discovery on spatial trajectories (i.e., finding the pair of most similar subtrajectories). Trajectory motifs are used in many applications, e.g., sports sense analysis [11], traffic analysis [15], or used as a building block for other trajectory mining and analysis methods [16, 31, 12]. As an example, Figure 1(a) visualizes a pedestrian's GPS trajectory from the GeoLife trajectory dataset [32], by a 3D plot with timestamp number at the horizontal axis. The motif corresponds to the most similar pair of subtrajectories (in red and blue). Figure 1(b) illustrates the motif (i.e., the two subtrajectories) on a map, which could be used in human behavior analysis.



(a) A pedestrian's trajectory April 10-12, 2009

(b) Discovered motif
red: 07:33-7:48, April 10, 2009
blue: 07:33-7:50, April 12, 2009

Figure 1: Subtrajectory motif discovered in a trajectory from the GeoLife trajectory dataset

It is important to choose a suitable similarity measure for motif discovery. The Fréchet metric is amongst the most popular measures for trajectory similarity [24, 10]. Generally speaking, the Fréchet distance between two spatial trajectories, S_a and S_b, is the length of the shortest leash needed to walk a dog when the person walks along S_a and the dog walks along S_b. In the geographic information handbook [10], the authors conclude that *"The most successful fundamental distance measure to this date is probably the Fréchet metric, which is one of the most natural measures to calculate the similarity between two trajectories"*. The Fréchet distance and its variants have been successfully used in a number of application domains, such as handwriting recognition [22], bioinformatics [27], computational geometry [5], as well as geographic applications [2]. In the literature, many recent systems have adopted the discrete Fréchet distance (DFD) to measure the distance between discrete trajectories (or the Fréchet distance for continuous curves) [2, 10, 3, 12, 25]. In addition, as we will elaborate in Section 2, DFD is particularly suitable for real-world spatial trajectories, which often exhibit the following properties: (i) non-uniform/varying sampling rate, and (ii) missing samples at some time points. For example, the GeoLife dataset [32], a real spatial trajectory dataset collected by Microsoft, has all the above properties.

In this paper, we discover motifs in spatial trajectories with DFD as the similarity measure. This problem is computationally challenging for two reasons:

(I) The computation of DFD between two subtrajectories takes $O(\ell^2)$ time [11], where ℓ denotes the subtrajectory length. There have been attempts to speed up DFD computation by using GPUs [12] or a faster algorithm (with $O(\ell^2 \cdot \frac{\log \log \ell}{\log \ell})$ time complexity) [1]. In contrast, we take an orthogonal research direction to reduce the number of DFD computations for motif discovery, e.g., by using various types of pruning on DFD computations and subtrajectory pairs.

(II) The problem involves $O(n^4)$ pairs of subtrajectories, where n is the length of the input trajectory/ies. The fact that DFD exhibits non-monotonicity (cf. Section 4.1) precludes us from applying efficient algorithmic paradigms (like binary search) to reduce the number of candidate pairs.

To overcome these challenges, we exploit the properties of DFD and devise lower bound functions that incur low computation time.

These lower bound functions serve for two purposes: first, pruning unpromising pairs of subtrajectories without invoking expensive DFD computation; second, guiding the search to discover the motif as soon as possible. Our lower bound functions are novel; nothing similar has been used in previous work. Additionally, they can be computed in amortized O(1) time.

Furthermore, we propose a grouping-based solution with multi-level pruning. This solution (i) divides the input trajectory into groups, (ii) prunes dissimilar pairs of groups, and then (iii) aggressively processes the surviving pairs of groups until the result is found. At the heart of this approach lies a suite of lower bound functions to prune unpromising pairs of groups.

All our techniques apply directly to motif discovery within the same or between different input trajectories. Importantly, besides motif discovery, they can be incorporated readily to other applications [2, 3] which employ DFD as the similarity measure.

2. RELATED WORK

In this section, we survey previous work but, due to the strict page limit, we focus only on the most relevant pieces. First, we present alternative similarity measures and pinpoint the advantages offered by the Fréchet metric that render it the ideal choice for (sub)trajectory similarity [10]. Next, we overview existing motif discovery approaches and juxtapose them to ours. Finally, we provide an outlook of other practically relevant trajectory analysis techniques.

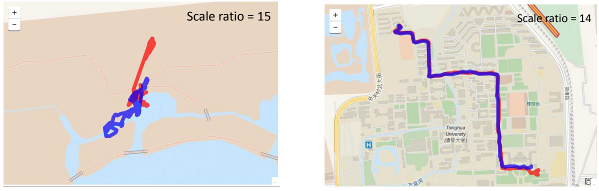
Trajectory similarity measures: Several similarity measures have been proposed for trajectories, e.g., Euclidean Distance (ED), Discrete Fréchet Distance (DFD) [8, 1], Dynamic Time Warping (DTW) [28], Longest Common Subsequence (LCSS) [26], Edit Distance on Real Sequence (EDR) [6]. Real-world trajectories (e.g., those in GeoLife dataset) exhibit two key characteristics, namely, non-uniform/varying sampling rate and missing samples for some time points. Thus, a desirable similarity measure would account for both these characteristics. In Table 1, we summarize the properties of the aforementioned trajectory similarity measures and their computation cost, expressed in terms of (sub)trajectory length ℓ. Local time shifting refers to the ability of tolerating short-term discrepancies (e.g., missing samples, measurement errors) in aligning two trajectories [6].

Distance metric	Non-uniform/varying sampling rate	Local time shifting	Computation cost
ED			$O(\ell)$
DTW		√	$O(\ell^2)$
LCSS		√	$O(\ell^2)$
EDR		√	$O(\ell^2)$
DFD	√	√	$O(\ell^2)$

Table 1: Distance measures and their characteristics

We will use examples to illustrate the advantages of discrete Fréchet Distance (DFD) over typical alternatives (e.g., ED, DTW). We first apply two different measures (ED and DFD) to compute motifs on the GeoLife trajectory dataset [32]. Figures 2(a) and 2(b) show the most similar pair of subtrajectories by ED and DFD, respectively. Observe that the result of DFD (in Figure 2(b)) captures much better a human's interpretation. The reason is that ED measures spatial proximity only, and dismisses the movement pattern.

In Figure 3, we demonstrate the effect of non-uniform sampling in real-world data using DTW and DFD between trajectories S_a, S_b and S_a, S_c. Trajectories S_a (black color) and S_b (blue color) are uniformly sampled, while trajectory S_c (red color) is non-uniformly sampled. Intuitively, trajectory S_c is more similar to S_a than S_b, i.e., DFD(S_a, S_c) < DFD(S_a, S_b), however, DTW(S_a, S_c) > DTW(S_a, S_b). The reason is DTW requires each point to be matched to another (and adds up all distances between matched pairs) thus being sensitive to non-uniform sampling.



(a) Most similar pair in ED
ED: 8.71 m; DFD: 0.09 m

(b) Most similar pair in DFD
ED: 19.42 m; DFD: 0.08 m

Figure 2: ED and DFD

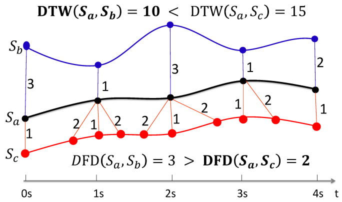

DTW(S_a, S_b) = 10 < DTW(S_a, S_c) = 15
S_b
S_a
S_c
3
1
3
1
2
1
2
1
2
2
1
2
1
2
1
DFD(S_a, S_b) = 3 > DFD(S_a, S_c) = 2
0s
1s
2s
3s
4s
t

Figure 3: DTW and DFD; S_c is non-uniformly sampled

In summary, ED is the fastest metric to compute but it is not robust to local time shifting. More robust measures, such as DTW [28], LCSS [26], EDR [6], are defined as the sum of point-to-point distances, which makes them sensitive to the sampling rate. As shown in Table 1, only DFD [8, 1], also known as the "dogman" distance, can tolerate non-uniform/varying sampling rate [11, 24, 12]. Other distance measures require that points along the trajectories are uniformly and densely sampled, which is rarely the case in real settings [11]. For more details on trajectory similarity measures, we refer the reader to surveys [10, 24, 7].

The parallel computing [12] and computational geometry [1] communities have proposed some techniques to speed up DFD computation. In contrast, in our work we take an orthogonal approach to accelerate DFD computations for trajectory motif discovery via novel pruning techniques.

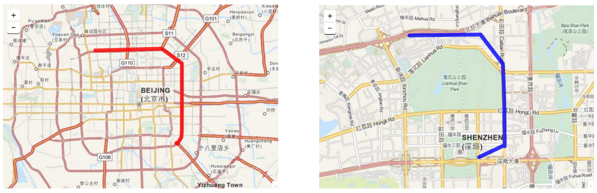
Trajectory motif discovery techniques: For spatial trajectories, most of the motif discovery techniques adopt the *symbolic approach* [17, 11, 20]. This approach employs symbols to represent pre-defined movement patterns; some example symbols and patterns are illustrated in Figure 4(a). To convert a trajectory into a string of symbols, it first partitions a trajectory into fragments, and then maps each fragment to a symbol (i.e., pre-defined movement pattern). After that, it applies substring matching techniques to discover motifs [30, 14]. Unfortunately, this approach may produce similar strings even if their original trajectories are far apart. For example, we illustrate two trajectories of Uber drivers (in two different cities) in Figures 4(b) and 4(c). Although these two trajectories are geographically far apart (in two different cities), both of them are mapped to string 'RVLH'. Since this approach cannot capture the spatial distance between trajectories, we dismiss it.

Motifs have also been studied for time series data [19, 18]. However, these techniques are tailored to time series with Euclidean distance, and are not suitable for spatial trajectories with DFD.

Other trajectory analysis techniques: Besides motif discovery, there are many other spatial trajectory analysis problems, e.g., convoy discovery [13], outlier detection [29, 17], trajectory cluster-

symbol	movement pattern
V	vertical long straight
H	horizontal long straight
L	left turn
R	right turn

(a) Pre-defined movement patterns and symbols



(b) A trajectory in Beijing; string: RVLH

(c) A trajectory in Shenzhen; string: RVLH

Figure 4: Example of the symbolic approach

ing [16, 11, 12], etc. We refer the interested reader to a recent survey [31].

3. PROBLEM STATEMENT

In this section, we introduce the problem and present a baseline solution, starting with several basic definitions.

DEFINITION 1 (SPATIAL TRAJECTORY & SUBTRAJECTORY). *A spatial trajectory $\mathcal{S} = \langle \cdots, s_i, \cdots \rangle$ is a sequence of points. We denote its trajectory length by $n = |\mathcal{S}|$.*

Given a trajectory $\mathcal{S}$, we denote a subtrajectory of $\mathcal{S}$ as $\mathcal{S}_{i,i_e} = \mathcal{S}[i..i_e]$, where $0 \leq i < i_e \leq n-1$.

Let $T(\mathcal{S}) = \langle \cdots, t_i, \cdots \rangle$ be a sequence of ascending timestamps, where t_i is the timestamp of location s_i in $\mathcal{S}$. The timestamps may be non-uniform.

We assume each point s_i is a latitude-longitude (φ_i, λ_i) pair. We measure the *ground distance* between two trajectory points $s_i = (\varphi_i, \lambda_i), s_j = (\varphi_j, \lambda_j)$ as the great circle distance on Earth [21]:

$$d_G(i,j) = 2R \arcsin \sqrt{\sin^2\left(\frac{\varphi_j - \varphi_i}{2}\right) + \cos\varphi_i \cos\varphi_j \sin^2\left(\frac{\lambda_j - \lambda_i}{2}\right)}$$

where R is the radius of the earth. Nevertheless, our methods are directly applicable to higher dimensions (e.g., 3-d data points) and other types of ground distance (e.g., Euclidean).

As discussed in Section 2, we adopt the discrete Fréchet distance (DFD) to measure the distance between two subtrajectories $\mathcal{S}_{i,i_e}$ and $\mathcal{S}_{j,j_e}$, defined as:

$$d_F(i,i_e,j,j_e) = \max \begin{cases} d_G(i_e,j_e) \\ \min \begin{cases} d_F(i,i_e-1,j,j_e), \\ d_F(i,i_e,j,j_e-1), \\ d_F(i,i_e-1,j,j_e-1) \end{cases} \end{cases}$$

For $i_e = i$ and $j_e = j$, $d_F(i,i,j,j) = d_G(i,j)$, and the DFD computation recursion terminates at $i_e = i$ and $j_e = j$.

We study the motif discovery problem within a single input trajectory or between different trajectories; for simplicity, we focus presentation on single input trajectory but also elaborate on (and evaluate) the latter variant too. To produce a meaningful trajectory motif $(\mathcal{S}_{i,i_e}, \mathcal{S}_{j,j_e})$, we require that: (i) subtrajectories $\mathcal{S}_{i,i_e}$ and $\mathcal{S}_{j,j_e}$ are sufficiently long (e.g., each has length at least ξ), and (ii) their timestamp intervals do not overlap.

PROBLEM 1 (TRAJECTORY MOTIF DISCOVERY PROBLEM). *Given a trajectory $\mathcal{S}$ and a minimum motif length ξ, return the pair of subtrajectories $\mathcal{S}_{i,i_e}$ and $\mathcal{S}_{j,j_e}$ with the smallest DFD distance $d_F(i,i_e,j,j_e)$ among all pairs of non-overlapping subtrajectories (that is, $i < i_e < j < j_e$) with length at least ξ (that is, $i_e > i+\xi, j_e > j+\xi$).*

As mentioned previously, a variant of Problem 1 is to discover a motif between different trajectories. I.e., considering two trajectories $\mathcal{S}$ and $\mathcal{T}$, to return the pair of subtrajectories $\mathcal{S}_{i,i_e}$ and $\mathcal{T}_{j,j_e}$ whose DFD is the smallest among all possible pairs of their subtrajectories.

With Problem 1 in mind, a straightforward solution is to enumerate all pairs of subtrajectories $(\mathcal{S}_{i,i_e}, \mathcal{S}_{j,j_e})$ and then compute the DFD value for each pair. Its time complexity is $O(n^6)$, as there are $O(n^4)$ pairs of subtrajectories and each call to DFD takes $O(\ell^2) = O(n^2)$ time. Even if we implement each call to DFD by [1], the time complexity is still $O(n^6 \cdot \frac{\log\log n}{\log n})$.

We observe that, for all subtrajectory pairs $(\mathcal{S}_{i,i_e}, \mathcal{S}_{j,j_e})$ with the same start point (i,j), their DFD computation can be shared via dynamic programming. By incorporating this idea into the above solution, we obtain BruteDP (Algorithm 1) – a brute force algorithm that uses dynamic programming. A further optimization is to eliminate redundant calls of the ground distance function $d_G(\cdot,\cdot)$. We propose to precompute all pairs of ground distances, and store them in matrix $d_G[\cdot][\cdot]$ for quick access.

Algorithm 1 BruteDP (Trajectory $\mathcal{S}$, minimum length ξ)

```
Input: trajectory $\mathcal{S}$, length $n$, minimum motif length $\xi$
Output: subtrajectory pair $bpair = (\mathcal{S}_{i,i_e}, \mathcal{S}_{j,j_e})$
1:  $bsf \leftarrow +\infty$; $bpair \leftarrow \emptyset$
2:  for $i \leftarrow 0$ to $n-2\xi+1$ do
3:      for $j \leftarrow i+\xi$ to $n-\xi+1$ do
4:          $d_F[i][j] \leftarrow d_G(i,j)$                          ▷ initialization
5:          for $t \leftarrow i+1$ to $n$ do
6:              $d_F[i][t] \leftarrow \max(d_G(i,t), d_F[i][t-1])$
7:              $d_F[t][j] \leftarrow \max(d_G(t,j), d_F[t-1][j])$
8:          for $i_e \leftarrow i+1$ to $j-1$ do                 ▷ share DFD computation
9:              for $j_e \leftarrow j+1$ to $n$ do
10:                 $tmp \leftarrow \min(d_F[i_e-1][j_e-1], d_F[i_e][j_e-1], d_F[i_e-1][j_e])$
11:                 $d_F[i_e][j_e] \leftarrow \max(d_G(i_e,j_e), tmp)$
12:                 if $i_e > i+\xi, j_e > j+\xi$ and $d_F[i_e][j_e] < bsf$ then
13:                     $bsf \leftarrow d_F[i_e][j_e]$; $bpair \leftarrow (\mathcal{S}_{i,i_e}, \mathcal{S}_{j,j_e})$
14: return $bpair$
```

Algorithm 1 can be adapted to motif discovery between different trajectories easily, i.e., with $\mathcal{S}_{j,j_e}$ playing the role of a subtrajectory in the second input trajectory, and by incrementing i until $n-\xi+1$ (instead of $n-2\xi+1$) at Line 2, and j starting from 0 (instead of $i+\xi$) at Line 3 (because this variant considers separate trajectories, thus not imposing the constraint $i < i_e < j < j_e$).

Analysis: With all pairs of ground distances available in matrix d_G, the time complexity of Algorithm 1 is $O(n^4)$, which is attributed to the nested for-loops for variables i, j (at Lines 2-3) and variables i_e, j_e (at Lines 8-9). The space complexity of the algorithm is $O(n^2)$, as it employs two 2-dimensional matrices: (i) $d_F[\cdot][\cdot]$ for implementing dynamic programming, and (ii) $d_G[\cdot][\cdot]$ for holding all-pair ground distances.

Before we proceed to our advanced techniques, we summarize frequently used notation in Table 2.

4. BOUNDING-BASED SOLUTION

We first analyze the properties of DFD (in Section 4.1). Then we exploit these properties to devise novel lower bound functions for DFD (in Section 4.2). Our lower bounds can be computed in amortized $O(1)$ time, and guarantee no false negatives (in Section

Symbol	Meaning
$\mathcal{S}$	input trajectory
$\mathcal{S}[i]$	(φ_i, λ_i), the i^{th} point of $\mathcal{S}$
$\mathcal{S}_{i,i_e}$	the subtrajectory of $\mathcal{S}$ starting at $\mathcal{S}[i]$ and ending at $\mathcal{S}[i_e]$
n	the length of trajectory $\mathcal{S}$
ξ	the minimum motif length
$d_G(i,j)$	the ground distance between $\mathcal{S}[i]$ and $\mathcal{S}[j]$
$d_F(i,i_e,j,j_e)$	the DFD between subtrajectories $\mathcal{S}_{i,i_e}$ and $\mathcal{S}_{j,j_e}$

Table 2: Notation

4.3). Finally, we propose a bounding-based solution that applies our lower bound functions to prune unpromising pairs of trajectories and reduce the number of DFD computations (in Section 4.4).

4.1 Properties of DFD

4.1.1 Non-monotonicity

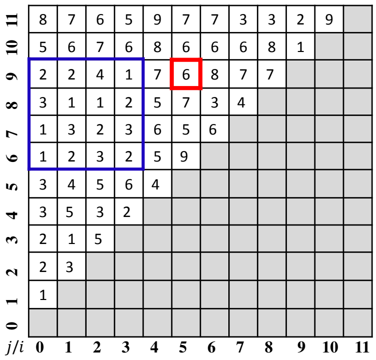
Typical sequence/string mining algorithms exploit the monotone property to develop efficient Apriori-style algorithms. An example of the monotone property would be: "given a string S, if q_α is a substring of q_β, then the frequency of q_α in S cannot be smaller than the frequency of q_β in S." It would be tempting to adapt such an idea to solve our problem efficiently. Unfortunately, the DFD metric does not satisfy the monotone property. Formally:

DEFINITION 2 (CONTAINMENT $\subseteq$). *$\mathcal{S}_{i,i_e}$ is said to contain $\mathcal{S}_{i',i'_e}$, denoted as $\mathcal{S}_{i',i'_e} \subseteq \mathcal{S}_{i,i_e}$, iff $i' \geq i$ and $i'_e \leq i_e$.*

LEMMA 1 (NON-MONOTONICITY). *Let $(\mathcal{S}_{i,i_e}, \mathcal{S}_{j,j_e})$ be a subtrajectory pair of S. Let $\mathcal{S}_{i',i'_e}, \mathcal{S}_{j',j'_e}$ be subtrajectories that satisfy $\mathcal{S}_{i',i'_e} \subseteq \mathcal{S}_{i,i_e}, \mathcal{S}_{j',j'_e} \subseteq \mathcal{S}_{j,j_e}$. It holds that, $d_F(i,i_e,j,j_e)$ is neither monotone increasing nor monotone decreasing with respect to $d_F(i',i'_e,j',j'_e)$.*

We provide a counter-example to demonstrate the non-monotonicity as follows.

Example: We illustrate Lemma 1 using a trajectory $\mathcal{S}$ with length $n = 12$. Figure 5 shows the ground distance for each pair $(\mathcal{S}[i], \mathcal{S}[j])$. Consider three subtrajectories $\mathcal{S}_{0,2} \subseteq \mathcal{S}_{0,3} \subseteq \mathcal{S}_{0,4}$ and their DFD distances from $\mathcal{S}_{6,9}$. Using Algorithm 1, we can compute these DFD values: $d_F(0,2,6,9) = 4$, $d_F(0,3,6,9) = 1$, $d_F(0,4,6,9) = 7$. When comparing $\mathcal{S}_{0,2}$ and $\mathcal{S}_{0,3}$, the DFD value (from $\mathcal{S}_{6,9}$) decreases from 4 to 1. However, when comparing $\mathcal{S}_{0,3}$ and $\mathcal{S}_{0,4}$, the DFD value (from $\mathcal{S}_{6,9}$) increases from 1 to 7. I.e., DFD does not satisfy the monotone property.



11	8	7	6	5	9	7	7	3	3	2	9	
10	5	6	7	6	8	6	6	6	8	1		
9	2	2	4	1	7	6	8	7	7			
8	3	1	1	2	5	7	3	4				
7	1	3	2	3	6	5	6					
6	1	2	3	2	5	9						
5	3	4	5	6	4							
4	3	5	3	2								
3	2	1	5									
2	2	3										
1	1											
0												
j/i	0	1	2	3	4	5	6	7	8	9	10	11

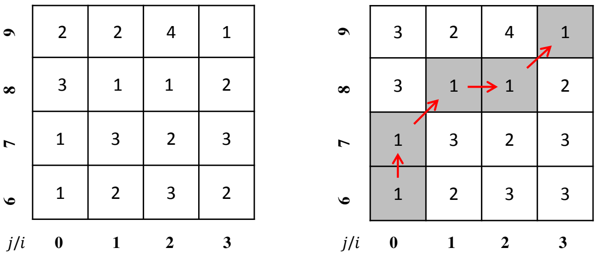
Figure 5: Example of d_G matrix

4.1.2 Crucial Observation

Non-monotonicity aside, we make a crucial observation which is quintessential to our approach. Specifically, the computation of DFD by recurrence is equivalent to a path finding problem in the d_G matrix.

OBSERVATION 1. *The DFD between $\mathcal{S}_{i,i_e}$ and $\mathcal{S}_{j,j_e}$ must be contributed by a path from (i,j) to (i_e,j_e) such that: (i) the path travels along non-decreasing positions, (ii) the worst-case ground distance along the path is minimized.*

We illustrate using two subtrajectories $\mathcal{S}_{0,3}$ and $\mathcal{S}_{6,9}$. Figure 6(a) shows the ground distance d_G for each pair of points from $\mathcal{S}_{0,3}$ and $\mathcal{S}_{6,9}$ (note that only the relevant part of the d_G matrix from Figure 5 is shown). We compute the d_F value for each pair of points, as illustrated in Figure 6(b). The DFD distance is $d_F(0,3,6,9) = 1$, which is contributed by the path of gray cells from $(0,6)$ to $(3,9)$, that minimizes the maximum ground distance among the cells it visits.



9	2	2	4	1
8	3	1	1	2
7	1	3	2	3
6	1	2	3	2
j/i	0	1	2	3

(a) Relevant part of d_G

9	3	2	4	1
8	3	1	1	2
7	1	3	2	3
6	1	2	3	3
j/i	0	1	2	3

(b) d_F computation as a path in d_G

Figure 6: DFD computation for $\mathcal{S}_{0,3}$ and $\mathcal{S}_{6,9}$

4.2 Pattern-based Lower Bounds

Based on Observation 1, we devise novel lower bound functions for DFD by accessing/traversing the d_G matrix according to different patterns (e.g., a single cell, cells in a cross, cells in a band).

Specifically, assuming that matrix d_G is precomputed and that bsf is the DFD of the best subtrajectory pair encountered so far in the search process, we propose a set of lower bound functions that apply to candidate subtrajectory pairs, or entire groups of candidate pairs, such that if the bound is greater than bsf, the candidates are safe to prune, i.e., to disqualify without further consideration, because they are guaranteed not to be the motif.

4.2.1 Cell-based Lower Bound

We refer to a subtrajectory pair $(\mathcal{S}_{i,i_e}, \mathcal{S}_{j,j_e})$ as candidate (i,i_e,j,j_e). We define a candidate subset $CS_{i,j}$ to represent all candidates with the same start positions i and j. This compact notation, using a pair (i,j), allows us to represent $O(n^2)$ candidates.

DEFINITION 3 (CANDIDATE SUBSET). *Given two start positions i and j, the candidate subset is defined as $CS_{i,j} = \{(i,i_e,j,j_e) : i_e > i \wedge j_e > j\}$.*

The following holds for any $CS_{i,j}$.

OBSERVATION 2. *For every $(i,i_e,j,j_e) \in CS_{i,j}$, the path leading to $d_F(i,i_e,j,j_e)$ must start from cell (i,j).*

For example, in Figure 6(a), for each candidate in $CS_{i,j}$, the path leading to DFD must start at cell $(0,6)$. We thus derive our first bound, which applies to any candidate in $CS_{i,j}$:

$$LB_{cell}(i,j) = d_G(i,j) \quad (1)$$

For every $(i, i_e, j, j_e) \in CS_{i,j}$, $LB_{cell}(i,j) \le d_F(i, i_e, j, j_e)$.

Example: In Figure 5, for candidate subset $CS_{5,9}$ (i.e., for all candidate pairs that start at the red cell), we obtain $LB_{cell}(5,9) = d_G(5,9) = 6$. This is a lower bound for the DFD of any candidate pair in $CS_{5,9}$. E.g., for pair $(\mathcal{S}_{5,6}, \mathcal{S}_{9,11})$, the exact DFD is $d_F(5,6,9,11) = 7$.

4.2.2 Cross-based Lower Bound

If a candidate subset is not pruned using LB_{cell}, we attempt to prune it with tighter lower bounds.

OBSERVATION 3. *For every $(i, i_e, j, j_e) \in CS_{i,j}$, the path leading to $d_F(i, i_e, j, j_e)$ must pass through the $(i+1)$-th column and $(j+1)$-th row.*

Example: In Figure 5, consider candidate $(4,6,8,10)$ in the candidate subset $CS_{4,8}$. For this candidate, any path from the start-cell (4,8) to the end-cell (6,10) must pass through the 5-th column and 9-th row; otherwise, the path cannot reach the end-cell (6,10). We thus define the following lower bounds.

$$LB_{row}(i,j) = \min_{i' \in [i, j-1]} \{d_G(i', j+1)\} \qquad (2)$$

$$LB_{col}(i,j) = \min_{j' \in [j, n-1]} \{d_G(i+1, j')\} \qquad (3)$$

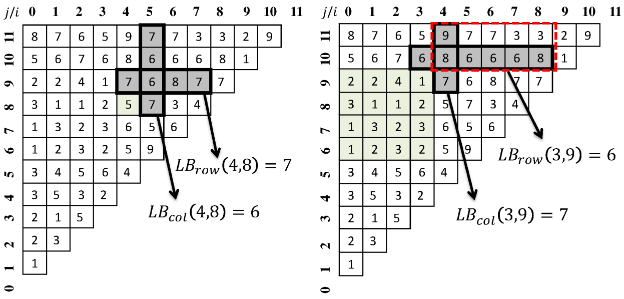
For every $(i, i_e, j, j_e) \in CS_{i,j}$, it holds that $LB_{row}(i,j) \le d_F(i, i_e, j, j_e)$ and that $LB_{col}(i,j) \le d_F(i, i_e, j, j_e)$. Thus, we combine the two into the cross-based lower bound below:

$$LB^{start}_{cross}(i,j) = \max\left(LB_{row}(i,j), LB_{col}(i,j)\right) \qquad (4)$$

For every $(i, i_c, j, j_c) \in CS_{i,j}$, $LB^{start}_{cross}(i,j) \le d_F(i, i_c, j, j_c)$.

Example: Consider cell (4,8) in Figure 7(a), and assume that $n = 12$. $LB^{start}_{cross}(4,8)$ is computed over the gray cells as follows:

$$\begin{aligned} LB^{start}_{cross}(4,8) &= \max(LB_{row}(4,8), LB_{col}(4,8)) \\ &= \max\left(\min_{i' \in [4,7]}\{d_G(i',9)\}, \min_{j' \in [8,11]}\{d_G(5,j')\}\right) \\ &= \max(6,6) = 6 \end{aligned}$$



(a) $LB^{start}_{cross}(4,8)$ (b) $LB^{end}_{cross}(3,9)$

Figure 7: Examples of cross-based bounds

4.2.3 Band-based Lower Bound

Our problem definition considers only subtrajectories with length at least ξ. Based on that, we extend Observation 3 to:

OBSERVATION 4. *For every $(i, i_e, j, j_e) \in CS_{i,j}$ that satisfies the constraint $i_e > i + \xi$ and $j_e > j + \xi$, the path leading to $d_F(i, i_e, j, j_e)$ must pass through columns $i+1$ to $i+\xi$ and through rows $j+1$ to $j+\xi$.*

Hence, we define the following band-based lower bounds:

$$LB^{row}_{band}(i,j) = \max_{j' \in [j, j+\xi-1]} \{LB_{row}(i,j')\} \qquad (5)$$

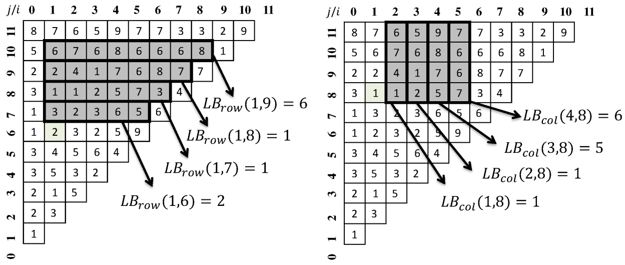
$$LB^{col}_{band}(i,j) = \max_{i' \in [i, i+\xi-1]} \{LB_{col}(i',j)\} \qquad (6)$$

For every $(i, i_e, j, j_e) \in CS_{i,j}$ where $i_e > i + \xi$ and $j_e > j + \xi$ it holds that:

$$LB^{row}_{band}(i,j) \le d_F(i, i_e, j, j_e) \qquad (7)$$

$$\text{and} \quad LB^{col}_{band}(i,j) \le d_F(i, i_e, j, j_e) \qquad (8)$$

If $LB^{row}_{band}(i,j) \ge bsf$ or $LB^{col}_{band}(i,j) \ge bsf$ we can safely prune $CS_{i,j}$.



(a) $LB^{row}_{band}(1,6)$ (b) $LB^{col}_{band}(1,8)$

Figure 8: Example of band-based bound

Example: Consider candidate subset $CS_{1,6}$ in Figure 8(a). Suppose the minimum motif length is $\xi = 4$ and $n = 12$. By the definition of $LB_{row}(i,j)$, the minimum values in the 7-th, 8-th, 9-th and 10-th row are 2, 1, 1 and 6, respectively. Hence, $LB^{row}_{band}(1,6) = \max(2,1,1,6) = 6$. Similarly, consider candidate subset $CS_{1,8}$ in Figure 8(b). By the definition of $LB_{col}(i,j)$, the minimum value of the 2-nd, 3-rd, 4-th and 5-th column are 1, 1, 5 and 6, respectively, as shown in Figure 8(b). Hence, $LB^{col}_{band}(1,8) = \max(1,1,5,6) = 6$.

4.2.4 Pruning within Candidate Subset

The bounds presented so far prune entire candidate subsets. If a candidate subset $CS_{i,j}$ survives these bounds, we need to consider candidate pairs inside of it. To avoid considering all candidate pairs in $CS_{i,j}$, here we introduce a cross-based bound that prunes candidate pairs within $CS_{i,j}$.

As introduced in Section 3, for all candidate pairs (i.e., $\mathcal{S}_{i,i_e}, \mathcal{S}_{j,j_e}$) in candidate set $CS_{i,j}$, their DFD computation can be shared via dynamic programming. Assume that at some point, the dynamic programming reaches end-cell (i_e, j_e), where $i_e - i > \xi$, $j_e - j > \xi$ and $bsf = d_F(i, i_e, j, j_e)$. We define the following cross-based lower bound for the end-cell:

$$LB^{end}_{cross}(i_e, j_e) = \max\left(LB_{row}(i_e, j_e), LB_{col}(i_e, j_e)\right) \qquad (9)$$

If (i, i_c, j, j_c) is a candidate in $CS_{i,j}$ where $i_c > i_e$ and $j_c > j_e$, it holds that $LB^{end}_{cross}(i_e, j_e) \le d_F(i, i_c, j, j_c)$. Hence, if $LB^{end}_{cross}(i_e, j_e) \ge bsf$, we can safely avoid expanding cell (i_e, j_e), i.e., eliminate paths within $CS_{i,j}$ that pass via cell (i_e, j_e).

Example: In Figure 7(b), suppose $\xi = 2$, $i = 0$, $j = 6$, $i_e = 3$

and $j_e = 9$. $LB^{end}_{cross}(i_e, j_e)$ is computed over the gray cells:

$$\begin{aligned} LB^{end}_{cross}(3,9) &= \max(LB_{row}(3,9), LB_{col}(3,9)) \\ &= \max\Big(\min_{i'_e \in [3,8]}\{d_G(i'_e, 10)\}, \min_{j'_e \in [9,11]}\{d_G(4, j'_e)\}\Big) \\ &= \max(6,7) = 7 \end{aligned}$$

If $LB^{end}_{cross}(3,9) \geq bsf$, we prune the candidates (i.e., subtrajectory pairs) in $CS_{0,6}$ whose end-cells fall in the red dotted box.

4.3 Relaxed Lower Bounds

If we follow the aforementioned equations directly, a cross-based bound takes $O(n)$ time to compute and a band-based bound takes $O(\xi n)$ time. Although both of them are more efficient than raw DFD computation (i.e., $O(n^2)$), in this section, we drop their amortized time complexity to $O(1)$ by relaxing them slightly. These relaxed bounds incur no false negatives, i.e., they are guaranteed not to miss the motif. Due to the space limit, we illustrate our relaxation approach for band-based bounds only. The relaxation of cross-based bounds follows the same lines.

The key idea is to employ one parameter per bound, and keep them in matrices for rapid access. First, we compute the minimum value for each column i and each row j:

$$C_{min}[i] \quad = \quad \min_{j' \in [0, j-1]}(d_G(i+1, j')) \qquad (10)$$

$$R_{min}[j] \quad = \quad \min_{i' \in [i, n-1]}(d_G(i', j+1)) \qquad (11)$$

This step takes $O(2 \cdot n \cdot n) = O(n^2)$ time.

We define the relaxed version of cross-based bounds as:

$$rLB^{start}_{cross}(i,j) \quad = \quad \max\{C_{min}[i], R_{min}[j]\} \qquad (12)$$

$$rLB^{end}_{cross}(i_e, j_e) \quad = \quad \max\{C_{min}[i_e], R_{min}[j_e]\} \qquad (13)$$

In turn, the relaxed band-based bounds are defined as:

$$rLB^{row}_{band}(j) \quad = \quad \max_{j' \in [j, j+\xi-1]}\{R_{min}[j']\} \qquad (14)$$

$$rLB^{col}_{band}(i) \quad = \quad \max_{i' \in [i, i+\xi-1]}\{C_{min}[i']\} \qquad (15)$$

We compute the relaxed version of cross-based bounds by computing $C_{min}[i]$ and $R_{min}[j]$ for each column i and each row j. This step takes $O(n)$ time per column/row. Similarly, we compute relaxed band-based bounds for each column i and each row j. This step takes $O(\xi n)$ time per row/column. Thus, the total computation time of cross-based and band-based lower bounds is $O(n \cdot n) = O(n^2)$ and $O(\xi n \cdot n) = O(n^2)$, respectively. By amortizing the computation time over all candidate subsets $CS_{i,j}$ (i.e., $O(n^2)$ of them), the computation time per $CS_{i,j}$ for each relaxed bound is only $O(n^2/n^2) = O(1)$.

The following lemma proves the correctness of the relaxed band-based bounds. The proof for the relaxed cross-based bounds follows the same lines and is omitted for brevity.

LEMMA 2. *It holds that:*

$$rLB^{row}_{band}(j) \leq LB^{row}_{band}(i,j) \quad \textit{and} \quad rLB^{col}_{band}(i) \leq LB^{col}_{band}(i,j)$$

PROOF.

$$\begin{aligned} &\min_{i' \in [0, j-1]}(d_G(i', j+1)) \leq \min_{i' \in [i, j-1]}(d_G(i', j+1)) \\ &\Rightarrow R_{min}[j] \leq LB_{row}(i,j) \\ &\Rightarrow \max_{j' \in [j, j+\xi-1]}\{R_{min}[j']\} \leq \max_{j' \in [j, j+\xi-1]}\{LB_{row}(i, j')\} \\ &\Rightarrow rLB^{row}_{band}(j) = LB^{row}_{band}(i,j) \end{aligned}$$

Similarly, $rLB^{col}_{band}(i) \leq LB^{col}_{band}(i,j)$. □

In the experiments, we compare the effectiveness of the original bounds with the relaxed ones. We summarize the time requirements of all lower bounds in Table 3.

Lower bound	Time	Relaxed bound	Time
$LB_{cell}(i,j)$	$O(1)$		
$LB^{start}_{cross}(i,j)$	$O(n)$	$rLB^{start}_{cross}(i,j)$	$O(1)$
$LB^{end}_{cross}(i_e,j_e)$	$O(n)$	$rLB^{end}_{cross}(i_e,j_e)$	$O(1)$
$LB^{row}_{band}(i,j)$	$O(\xi n)$	$rLB^{row}_{band}(j)$	$O(1)$
$LB^{col}_{band}(i,j)$	$O(\xi n)$	$rLB^{col}_{band}(i)$	$O(1)$

Table 3: Summary of lower bounds

4.4 Optimized Solution

Combining all bounds: Given a candidate subset $CS_{i,j}$, we compute a tighter lower bound for $CS_{i,j}$, denoted by $CS_{i,j}.LB$, using:

$$\max\{LB_{cell}(i,j), rLB^{start}_{cross}(i,j), rLB^{row}_{band}(j), rLB^{col}_{band}(i)\}.$$

This lower bound takes $O(1)$ time because each term can be obtained in $O(1)$ time, as shown in Table 3.

Prioritizing search order: To support effective pruning of $CS_{i,j}$ by lower bounds, it is desirable to obtain a small bsf (i.e., a good temporary motif) as early as possible. Intuitively, a candidate subset with small $CS_{i,j}.LB$ tends to contain a candidate with small DFD value. Thus, we propose to process $CS_{i,j}$ in ascending order of $CS_{i,j}.LB$.

Putting it all together: Algorithm 2 presents the pseudocode for *bounding-based trajectory motif* (BTM), which incorporates all above ideas to solve the trajectory motif discovery problem.

Algorithm 2 BTM (Trajectory $\mathcal{S}$, minLength ξ)

```
    Input: trajectory $\mathcal{S}$, length $n$, minimum motif length $\xi$
    Output: subtrajectory pair $bpair = (\mathcal{S}_{i,i_e}, \mathcal{S}_{j,j_e})$
 1: $bsf \leftarrow +\infty$; $bpair \leftarrow \emptyset$; $j_{end} \leftarrow n$
 2: Compute { $LB_{cell}, rLB^{start}_{cross}, rLB^{end}_{cross}, rLB^{row}_{band}, rLB^{col}_{band}$ }
 3: Construct a list **A** with one element **a** per candidate subset
 4: Sort **A** in ascending order of $\mathbf{a}.LB$
 5: **for** each **a** in **A** with $bsf > \mathbf{a}.LB$ **do**
 6:     **for** $i_e \leftarrow \mathbf{a}.i + 1$ to $\mathbf{a}.j$ **do**
 7:         **for** $j_e \leftarrow \mathbf{a}.j + 1$ to $j_{end}$ **do**
 8:             $tmp \leftarrow \min(d_F[i_e\text{-}1][j_e\text{-}1], d_F[i_e][j_e\text{-}1], d_F[i_e\text{-}1][j_e])$
 9:             $d_F[i_e][j_e] \leftarrow \max(d_G(i_e, j_e), tmp)$
10:             **if** $i_e > \mathbf{a}.i + \xi, j_e > \mathbf{a}.j + \xi$ and $d_F[i_e][j_e] < bsf$ **then**
11:                 $bsf \leftarrow d_F[i_e][j_e]$; $bpair \leftarrow (\mathcal{S}_{i,i_e}, \mathcal{S}_{j,j_e})$
12:         **if** $bsf \leq rLB^{end}_{cross}(bpair.i_e, bpair.j_e)$ **then**
13:             $j_{end} \leftarrow bpair.j_e$      ▷ Pruning by $LB^{end}_{cross}(i_e, j_e)$ from Equation 9
14: return $bpair$
```

At Line 2, we first compute all lower bounds (and store them in matrices). Then, we insert each candidate subset $CS_{i,j}$ with its bound $CS_{i,j}.LB$ into a list (at Line 3), and sort that list (at Line 4). Next, we process the elements of the list in the sorted order. For each candidate subset, we examine its candidates via nested loops (at Lines 6-7), and compute the DFD of each candidate (at Line 8-9). Finally, we update bsf and the temporary motif pair (at Lines 10-11). Note that Lines 12-13 implement pruning by $LB^{end}_{cross}(i_e, j_e)$, as defined in Equation 9; this essentially performs pruning within the candidate subset currently considered, by disqualifying some of the candidate pairs it contains.

The lower bounds presented in this section are also applicable to motif discovery between different trajectories. Hence, similarly to Algorithm 1, Algorithm 2 is readily applicable to that problem variant too.

Analysis: The time complexity of Algorithm 2 is $O(n^4)$ in the worst case, which is attributed to the nested for-loops for variables **a** [that is, $O(n^2)$ iterations], i_e [that is, $O(n)$ iterations] and j_e [that is, $O(n)$ iterations] at Lines 5-7. The space complexity of Algorithm 2 is $O(n^2)$.

Algorithm 2 follows the best-first search paradigm with several effective lower bounds. As we will show in the experimental evaluation, it outperforms Algorithm 1 by two orders of magnitude.

5. GROUPING-BASED SOLUTION

In this section, we enhance the scalability of our techniques for long trajectories. Inspired by trajectory indexing methods [4, 9], we organize trajectory points into groups, then attempt pruning unpromising pairs of groups, before applying our solution from Section 4. To enable pruning, we design novel bounding functions for DFD on groups.

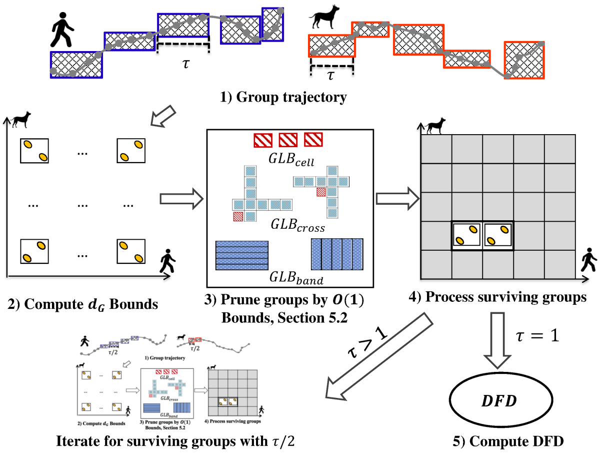


Figure 9: Grouping-based computation framework

We outline our grouping-based computation framework in Figure 9. First, we divide a trajectory into groups of τ samples (where τ is a tunable parameter), and compute a ground distance bound for each group pair (Steps 1 and 2, in Section 5.1). Next, we apply $O(1)$-time lower bounds (Step 3, in Section 5.2) to prune group pairs, before using tighter bounds for pruning (Step 4, in Section 5.3). For the surviving group pairs, we repeat the above steps by halving the group size, until τ reaches 1. Finally, we compute the exact DFD of candidates in the surviving groups (Step 5).

By combining the advantages of all techniques in Section 4 and in the current one, our grouping based computation framework outperforms the baseline solution by over 3 orders of magnitude. Importantly, all our techniques conduct only safe pruning, meaning that they produce exact answers (motifs).

5.1 Grouping Trajectory Points

We employ a group size parameter τ in order to partition a long trajectory into small groups. We proceed to define a group and the ground distances between groups.

DEFINITION 4 (τ-GROUPING). *Given the group size τ, we define the u-th group as the interval $g_u = [u\tau, (u+1)\tau - 1]$.*

For two groups g_u and g_v, we define the minimum and the maximum ground distance between them as:

$$d_G^{min}(g_u, g_v) = \min_{i \in g_u, j \in g_v} d_G(i, j) \tag{16}$$

$$d_G^{max}(g_u, g_v) = \max_{i \in g_u, j \in g_v} d_G(i, j) \tag{17}$$

By Definition 4, the ground distances between two groups satisfy the following property:

COROLLARY 1. *For every $i \in g_u$, $j \in g_v$, it holds that:*

$$d_G^{min}(g_u, g_v) \le d_G(i, j) \le d_G^{max}(g_u, g_v)$$

We utilize this property to devise lower bound functions in Sections 5.2, 5.3.

Example: Consider a trajectory $\mathcal{S}$ with $n = 12$ points. Given $\tau = 2$, we obtain six groups: $g_0, g_1, g_2, g_3, g_4, g_5$, as illustrated in Figure 10(a). For example, for groups $g_2 = [4, 5]$ and $g_5 = [10, 11]$, we compute the minimum ground distance as $d_G^{min}(g_2, g_5) = \min(d_G(4, 10), d_G(4, 11), d_G(5, 10), d_G(5, 11)) = 6$, and the maximum ground distance as $d_G^{max}(g_2, g_5) = \max(8, 9, 6, 7) = 9$. We show the minimum and maximum ground distances for group pair g_2 and g_5 in Figure 10(b).

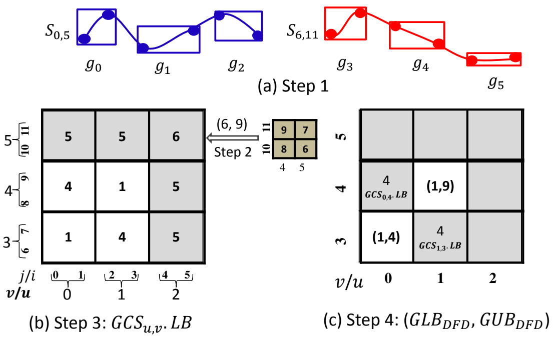


Figure 10: Example of 2-grouped trajectory

5.2 Pattern-based Bounds for Groups

To enable pruning on unpromising pairs of groups, we adapt our proposed lower bounds in Section 4 to groups. We denote the corresponding lower bounds with prefix $\mathcal{G}$, i.e., $\mathcal{G}LB_{cell}(u, v)$, $\mathcal{G}LB_{cross}^{start}(u, v)$, $\mathcal{G}LB_{cross}^{end}(u_e, v_e)$, $\mathcal{G}LB_{band}^{row}(u, v)$, and $\mathcal{G}LB_{band}^{col}(u, v)$. Later, we discuss their $O(1)$-time implementation.

Cell-based lower bound: We first define the cell-based lower bound for groups, denoted by $\mathcal{G}LB_{cell}$, as follows:

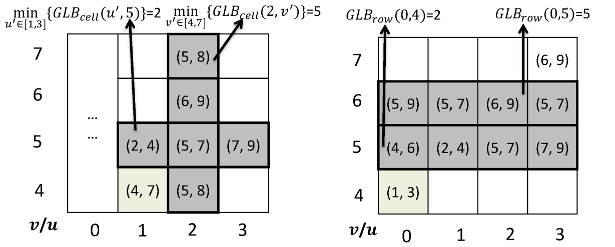
$$\mathcal{G}LB_{cell}(u, v) = d_G^{min}(g_u, g_v) \tag{18}$$

In Figure 10(b), $\mathcal{G}LB_{cell}(2, 5) = d_G^{min}(2, 5) = 6$. For any $i \in u$ and $j \in v$, it holds that $\mathcal{G}LB_{cell}(u, v) \le d_F(i, i_e, j, j_e)$.

Cross-based lower bounds: Next, we show that the cross-based lower bounds for groups can be expressed in terms of $\mathcal{G}LB_{cell}(u, v)$. We demonstrate using an example, rather than presenting ugly definitions and lemmas.

We denote the row and column based lower bounds for groups as $\mathcal{G}LB_{row}(u, v)$ and $\mathcal{G}LB_{col}(u, v)$, respectively. In Figure 11(a), assuming $n = 16$ and $\tau = 2$, we obtain $\mathcal{G}LB_{row}(1, 4) = \min_{u' \in [1,3]}(\mathcal{G}LB_{cell}(u', 5)) = \min(2, 5, 7) = 2$. Similarly, $\mathcal{G}LB_{col}(1, 4) = \min(5, 5, 6, 5) = 5$. The cross-based lower bound for start-cell (1,4) is $\mathcal{G}LB_{cross}^{start}(1, 4) =$

$\max(\mathcal{G}LB_{row}(1,4), \mathcal{G}LB_{col}(1,4)) = \max(2,5) = 5$. $\mathcal{G}LB^{end}_{cross}(u_e, v_e)$ is defined similarly to $\mathcal{G}LB^{start}_{cross}(u,v)$.



(a) $\mathcal{G}LB^{start}_{cross}(1,4)$ (b) $\mathcal{G}LB^{row}_{band}(0,4)$

Figure 11: Grouping based lower bounds

Band-based lower bounds: We also present band-based lower bounds for groups using an example. In Figure 11(b), we illustrate $\mathcal{G}LB^{row}_{band}(0,4)$. We compute it as $\mathcal{G}LB^{row}_{band}(0,4) = \max_{v'\in[4,5]}(\mathcal{G}LB_{row}(0,v')) = \max(2,5) = 5$. $\mathcal{G}LB^{col}_{band}(u,v)$ is defined similarly to $\mathcal{G}LB^{row}_{band}(u,v)$.

Relaxed lower bounds for groups: The concept of relaxed lower bounds, introduced in Section 4.3, can be adapted directly to the above pattern-based bounds for groups. This allows us to obtain relaxed lower bounds for groups in $O(1)$ time.

5.3 Bounding by DFD Computation

By exploiting the recurrence of DFD, we devise a tighter lower bound and a tighter upper bound for pairs of groups. While the lower bound is used to prune unpromising pairs of groups, the upper bound can be used to tighten bsf and thus improve the effectiveness of pruning.

Below we define a subtrajectory group, together with group-based DFD bounds.

DEFINITION 5 (GROUP-BASED DFD). *Let subtrajectory group $\mathcal{G}_{t,t_e}$ correspond to the interval $[t\tau, (t_e+1)\tau - 1]$, i.e., it covers group t to group t_e.*

Given two subtrajectory groups $\mathcal{G}_{u,u_e}$ and $\mathcal{G}_{v,v_e}$, we define the group-based DFD bounds $d_{F\,min}(u,u_e,v,v_e)$ and $d_{F\,max}(u,u_e,v,v_e)$ as:

$$d_{F\,min}(u,u_e,v,v_e) = \max\begin{cases} d_G^{min}(g_{u_e}, g_{v_e}) \\ \min\begin{cases} d_{F\,min}(u,u_e-1,v,v_e) \\ d_{F\,min}(u,u_e-1,v,v_e-1) \\ d_{F\,min}(u,u_e,v,v_e-1)\end{cases}\end{cases}$$

$$d_{F\,max}(u,u_e,v,v_e) = \max\begin{cases} d_G^{max}(g_{u_e}, g_{v_e}) \\ \min\begin{cases} d_{F\,max}(u,u_e-1,v,v_e) \\ d_{F\,max}(u,u_e-1,v,v_e-1) \\ d_{F\,max}(u,u_e,v,v_e-1)\end{cases}\end{cases}$$

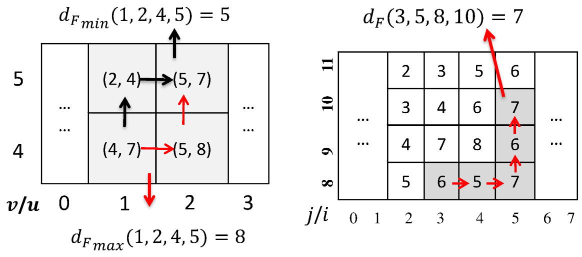
The following lemma proves the bounding property of $d_{F\,min}(u,u_e,v,v_e)$ and $d_{F\,max}(u,u_e,v,v_e)$.

LEMMA 3. *Let $\mathcal{G}_{u,u_e}$ and $\mathcal{G}_{v,v_e}$ be two subtrajectory groups. If a pair of subtrajectories $\mathcal{S}_{i,i_e}, \mathcal{S}_{j,j_e}$ satisfies $i \in g_u, j \in g_v, i_e \in g_{u_e}$ and $j_e \in g_{v_e}$, it holds that:*

$$d_{F\,min}(u,u_e,v,v_e) \le d_F(i,i_e,j,j_e) \le d_{F\,max}(u,u_e,v,v_e)$$

PROOF. $\forall u' \in [u,u_e], \forall v' \in [v,v_e]$ and $i' \in g_{u'}, j' \in g_{v'}$, according to Corollary 1, we have $d_G^{min}(g_{u'}, g_{v'}) \le d_G(i',j')$. By Observation 1, $d_{F\,min}(u,u_e,v,v_e)$ is attributed to a path among $d_G^{min}(g_{u'}, g_{v'})$ values, and $d_F(i,i_e,j,j_e)$ to a path among $d_G(i',j')$ values. Hence, for $i \in g_u, j \in g_v, i_e \in g_{u_e}, j_e \in g_{v_e}$, it holds that $d_{F\,min}(u,u_e,v,v_e) \le d_F(i,i_e,j,j_e)$. $d_{F\,max}(u,u_e,v,v_e) \ge d_F(i,i_e,j,j_e)$ is proven similarly. □

Example: Consider two subtrajectory groups $\mathcal{G}_{1,2}$ and $\mathcal{G}_{4,5}$ in Figure 12(a) and assume that $n = 12$. Their DFD bounds are $d_{F\,min}(1,2,4,5) = 5$ and $d_{F\,max}(1,2,4,5) = 8$, respectively. In Figure 12(b), the pair of subtrajectories $\mathcal{S}_{3,5}, \mathcal{S}_{8,10}$ has $d_F(3,5,8,10) = 7$. In accordance with Lemma 3, this distance falls indeed into range $[5,8]$.



(a) DFD bounds on groups (b) DFD on original trajectory

Figure 12: Illustration of DFD bounds

Recall that our problem definition enforces a minimum motif length ξ. To comply with it, we define the following lower and upper bounds between two groups g_u and g_v:

$$\begin{aligned} \mathcal{G}LB_{DFD}(u,v) &= \min_{u_e,v_e}\{d_{F\,min}(u,u_e,v,v_e) : \\ &\qquad u_e - u > \frac{\xi}{\tau} \wedge v_e - v > \frac{\xi}{\tau}\} \end{aligned} \quad (19)$$

$$\begin{aligned} \mathcal{G}UB_{DFD}(u,v) &= \min_{u_e,v_e}\{d_{F\,max}(u,u_e,v,v_e) : \\ &\qquad u_e - 1 - u > \frac{\xi}{\tau} \wedge v_e - 1 - v > \frac{\xi}{\tau}\} \end{aligned} \quad (20)$$

The following lemma shows their correctness. It is derived by applying the $\min_{u_e,v_e}$ function to both sides of Lemma 3.

LEMMA 4. *$\forall\, i \in g_u, \forall\, i \in g_v$, and $i_e > i + \xi, j_e > j + \xi$ it holds that:*

$$\mathcal{G}LB_{DFD}(u,v) \le d_F(i,i_e,j,j_e) \le \mathcal{G}UB_{DFD}(u,v)$$

$\mathcal{G}UB_{DFD}(u,v)$ allows us to tighten bsf, which in turn boosts the effectiveness of pruning. Both $\mathcal{G}LB_{DFD}(u,v)$ and $\mathcal{G}UB_{DFD}(u,v)$ can be computed in $O((\frac{n}{\tau})^2)$. We can reduce their computation cost by early termination. Specifically, if at some point during the computation of $\mathcal{G}LB_{DFD}(u,v)$, it holds that $\mathcal{G}LB^{end}_{cross}(u_e,v_e) \ge \mathcal{G}LB_{DFD}(u,v)$ with $u_e - u > \frac{\xi}{\tau} \wedge v_e - v > \frac{\xi}{\tau}$, we may safely terminate the computation because $\forall u_{e'} > u_e$ and $\forall v_{e'} > v_e$ it must be that $d_{F\,min}(u,u_{e'},v,v_{e'}) > d_{F\,min}(u,u_e,v,v_e)$ (i.e., it cannot further tighten the bound). Similarly, early termination is possible in the calculation of $\mathcal{G}UB_{DFD}(u,v)$ too.

5.4 GTM Algorithm

Algorithm 3 presents the pseudocode for *grouping-based trajectory motif* (GTM), which implements the computation framework depicted in Figure 9. We first construct groups at Line 3, then we compute the pattern-based lower bounds of group pairs at Lines 4-5. Next, we insert each grouping based candidate subset $\mathcal{G}CS_{u,v}$ with its bound $\mathcal{G}CS_{u,v}.LB = \max(\mathcal{G}LB_{cell}(u,v),$

$r\mathcal{G}LB^{start}_{cross}(u,v), r\mathcal{G}LB^{row}_{band}(u,v), r\mathcal{G}LB^{col}_{band}(u,v))$ into a list. Then, we process the list in ascending order of $\mathcal{G}CS_{u,v}.LB$, and apply DFD bounds for pruning (Lines 10-11) or for tightening the bsf (Lines 12-13). After that, we halve the group size and repeat the above procedure on the set of surviving groups $\mathcal{S}_{survive}$ until the group size drops to 1. When this happens (i.e., $\tau = 1$), each element in $\mathcal{S}_{survive}$ is a candidate subset $CS_{i,j}$. We invoke Algorithm 2 on $\mathcal{S}_{survive}$ to obtain the final result.

Algorithm 3 GTM (Trajectory $\mathcal{S}$, minLength ξ, group size τ)

```
    Input: trajectory $\mathcal{S}$, length $n$, minimum motif length $\xi$, group size $\tau$
    Output: subtrajectory pair $bpair = (\mathcal{S}_{i,i_e}, \mathcal{S}_{j,j_e})$
 1: $bsf \leftarrow +\infty$; $bpair \leftarrow \emptyset$
 2: while $\tau > 1$ do
 3:     Group trajectory $\mathcal{S}$ to $\mathcal{G}$                              ▷ Section 5.1
 4:     Compute $\mathcal{G}LB_{cell}, r\mathcal{G}LB^{start}_{cross}, r\mathcal{G}LB^{end}_{cross}$
 5:         and $r\mathcal{G}LB^{row}_{band}, r\mathcal{G}LB^{col}_{band}$                        ▷ Section 5.2
 6:     Construct a list $\mathcal{G}\mathbf{A}$ of candidate subsets
 7:     Sort $\mathcal{G}\mathbf{A}$ in ascending order of $\mathcal{G}\mathbf{a}.LB$
 8:     $\mathcal{S}_{survive} \leftarrow \emptyset$                          ▷ set of surviving groups
 9:     for each $\mathcal{G}\mathbf{a}$ in $\mathcal{G}\mathbf{A}$ with $bsf > \mathcal{G}\mathbf{a}.LB$ do           ▷ Section 5.3
10:         if $bsf > \mathcal{G}LB_{DFD}(\mathcal{G}\mathbf{a}.u, \mathcal{G}\mathbf{a}.v)$ then
11:             $\mathcal{S}_{survive} \leftarrow \mathcal{S}_{survive} \cup \mathcal{G}\mathbf{a}.u \cup \mathcal{G}\mathbf{a}.v$
12:         if $bsf > \mathcal{G}UB_{DFD}(\mathcal{G}\mathbf{a}.u, \mathcal{G}\mathbf{a}.v)$ then
13:             $bsf \leftarrow \mathcal{G}UB_{DFD}(\mathcal{G}\mathbf{a}.u, \mathcal{G}\mathbf{a}.v)$
14:     $\tau \leftarrow \tau/2, \mathcal{S} \leftarrow \mathcal{S}_{survive}$
15: Invoke Lines 5-13 in Alg. 2 on $\mathcal{S}_{survive}$ to compute the result $bpair$
```

Example: We demonstrate the grouping-based computation framework in Figure 10. Assume that the minimum trajectory motif length is $\xi = 2$ with $bsf = 5$. We first assign these subtrajectories into groups (with $\tau = 2$), as illustrated in Figure 10(a). Consider two subtrajectories $\mathcal{S}_{0,5}$ and $\mathcal{S}_{6,11}$. We compute $O(1)$-time pattern-based bounds to prune group pairs; the pruned pairs are shown in gray in Figure 10(b). Then, we compute $\mathcal{G}LB_{DFD}, \mathcal{G}UB_{DFD}$ bounds for surviving pairs, as illustrated in Figure 10(c). The upper bounds allow us to tighten bsf (to 4), whereas the lower bounds are used to prune pairs (i.e., the gray region in Figure 10(c)). Finally, we process the two surviving cells with Algorithm 2.

The group lower bounds developed in this section are directly applicable to motif discovery between different trajectories, and the adaptation of Algorithm 3 to that variant is straightforward.

Analysis: The computation cost of the while-loop (Lines 2–14) is $O(\sum_{i=1}^{log(\tau)} (\frac{c_i}{\tau})^4)$, where $c_1 = n$ and c_i is the number of surviving groups in iteration i. Line 16 takes $O(c\tau^2 n^2)$ time, where c is the number of surviving groups after the while-loop. In summary, the time complexity of Algorithm 3 is $O(\sum_{i=1}^{log(\tau)} (\frac{c_i}{\tau})^4 + c\tau^2 n^2)$. In the worst case, Algorithm 3 degenerates to Algorithm 2, with time complexity $O(n^4)$.

The space complexity of the algorithm is $O(n^2)$ as it employs two 2-dimensional matrices for precomputed ground distances (i.e., $d_G[\cdot][\cdot]$) and DFD values (i.e., $d_F[\cdot][\cdot]$). In addition, it takes $O((\frac{n}{\tau})^2)$ space for precomputed group based lower bounds in $\mathcal{G}\mathbf{A}$ at Line 6.

5.5 Space-efficient GTM: GTM*

We present a space-efficient variant of GTM, called GTM*. It incorporates three ideas: (i) during DFD computation, we compute ground distances on-the-fly, (ii) implement DFD computation with $O(n)$ space, and (iii) execute the while-loop only once for a given τ. Idea (i) eliminates the need for precomputed ground distances (i.e., $d_G[\cdot][\cdot]$). Idea (ii) is feasible because, in Lines 8-9 of Algorithm 2, we examine at most two rows of $d_F[\cdot][\cdot]$ at the same time. Idea (iii) requires only $O((\frac{n}{\tau})^2)$ space. Thus, the space complexity of GTM* is $O(\max\{(\frac{n}{\tau})^2, n\})$.

The time complexity of GTM* is $O((\frac{n}{\tau})^4 + c'\tau^2 n^2)$, where c' is the number of group pairs that survive pruning by Idea (i). Since GTM* executes the while-loop only once for a given τ (Idea iii), the value of c' in GTM* is expected to be larger than c in GTM.

6. EMPIRICAL EVALUATION

In this section, we evaluate the performance of our solutions on real data. Section 6.1 introduces the experimental setting. Section 6.2 studies the effectiveness of our pruning techniques (e.g., lower bounds and grouping). Section 6.3 compares the performance of different methods with respect to various parameters.

6.1 Experimental Setup

We used three real trajectory datasets from moving people, vehicles and animals. We note that these datasets have different characteristics (such as sampling frequency and data distribution) thus helping us verify the generality of our findings. The details of each dataset are as follows.

GeoLife[1]**:** This GPS trajectory dataset was collected in the GeoLife project by Microsoft. The trajectories were recorded by different GPS loggers and GPS-phones, and therefore they have different sampling rates. Each trajectory is a sequence of time-stamped points, each with a latitude, a longitude and an altitude. This dataset contains 17,621 trajectories with a total distance of 1.2 million kilometers.

Truck[2]**:** This dataset contains 276 trajectories of 50 trucks moving in Athens metropolitan area in Greece. The trucks were carrying concrete to several construction sites for 33 days.

Wild-Baboon[3]**:** This dataset was collected from wild olive baboons at Mpala Research Centre in Kenya [23]. It contains 25 trajectories of baboons with a custom-designed GPS collar that recorded a location every second from 1-st August to 14-th August, 2012.

In our experiments, we report the average measurements over 10 different trajectories of the same length. The response times reported include the precomputation time of distances and lower bounds. For each dataset, we concatenate raw trajectories in order to build longer trajectories. By default, we fix the motif length threshold ξ to 100, and the trajectory length n to 5000.

We used C++ for the implementation and conducted all experiments (with single thread) on a machine with an Intel Core i7- 4770 3.40GHz processor. We compare the following methods:

- the baseline solution BruteDP (cf. Algorithm 1)
- the bounding-based solution BTM (cf. Algorithm 2)
- the grouping-based solution GTM (cf. Algorithm 3)
- the space-efficient solution GTM* (cf. Section 5.5)

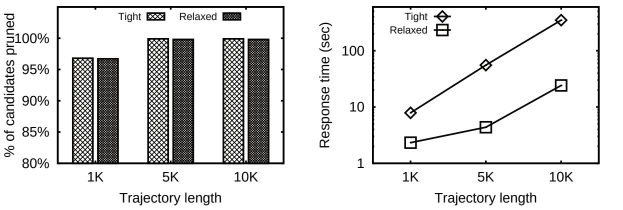
6.2 Pruning Effectiveness

We first assess the effectiveness of our pruning techniques, particularly of our lower bounds and grouping. For the purposes of this subsection, we present results only on the GeoLife dataset. Results on Truck and Wild-Baboon are similar and are omitted in the interest of space.

[1] http://research.microsoft.com/en-us/projects/geolife/default.aspx

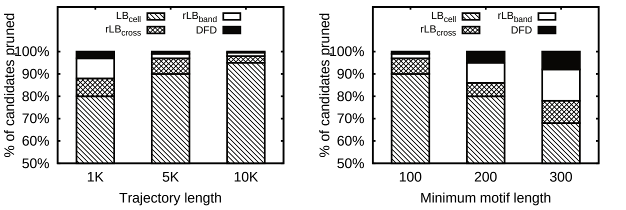
[2] http://chorochronos.datastories.org/

[3] https://www.datarepository.movebank.org/handle/10255/move.405



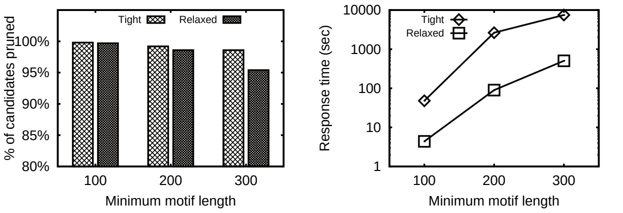
(a) Pruning ratio (b) Response time

Figure 13: BTM, effect of trajectory length n



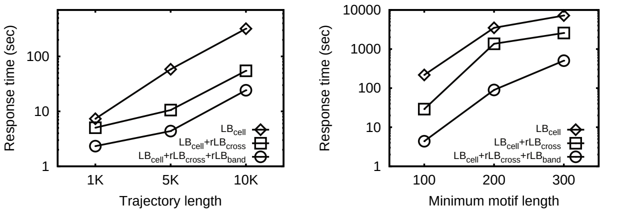
(a) Effect of trajectory length n (b) Effect of minimum motif length ξ

Figure 15: BTM, pruning ratio breakdown



(a) Pruning ratio (b) Response time

Figure 14: BTM, effect of minimum motif length ξ



(a) Effect of trajectory length n (b) Effect of minimum motif length ξ

Figure 16: BTM, response time

6.2.1 *Effectiveness of Relaxed Bounds*

We first compare two variants of BTM that use: (i) only the *tight lower bounds* from Section 4.2, and (ii) only the *relaxed lower bounds* from Section 4.3.

In Figure 13, we compare the tight with the relaxed bounds by varying the trajectory length n, with ξ fixed to 100. The pruning percentage in Figure 13(a) corresponds to the ratio of candidate pairs successfully pruned to the total number of candidate pairs. Note that because the percentage is high, and in order to show enough detail, we truncated the y-axis of the plot to start from 80%. In Figure 13(b), we show the overall response time to compute the motif. We observe that the relaxed bounds are only slightly weaker in pruning power, but they are orders of magnitude faster computation-wise.

In Figure 14, we investigate the effectiveness and performance of tight and relaxed bounds as a function of the minimum motif length ξ, with n fixed to 5000. Again, although the tight bounds have slightly higher pruning ratio (in Figure 14(a)), the relaxed bounds render motif computation 10 times faster (in Figure 14(b)). Since the relaxed bounds perform much better, we adopt them in our framework (instead of the tight ones) and use them in the subsequent experiments.

6.2.2 *Effectiveness of Lower Bounds*

In the next experiment, we compare the pruning effectiveness of the different lower bound functions ($LB_{cell}, rLB_{cross}, rLB_{band}$) using BTM. Each bar in Figure 15 corresponds to the total number of candidate pairs, broken down into the fraction pruned by each of the 3 types of bounds, and the fraction of the surviving pairs that required exact DFD computation (labeled as DFD in the bar charts). In Figures 15(a),(b) we vary the trajectory length n and the minimum motif length ξ, respectively. The bars are truncated to start at ratio 50% to retain detail, because the percentage of LB_{cell} hugely dominates the rest.

Over 92% of the candidates can be collectively pruned by our lower bounds. An interesting observation is that the bounds complement each other. For instance, when ξ increases (in Figure 15(b)), although LB_{cell} deteriorates, rLB_{band} becomes stronger, thus eliminating many of the candidates that survived LB_{cell}. This renders our methodology robust to different problem settings.

Next, we compare three variants of BTM that use: (i) LB_{cell} only, (ii) LB_{cell}, rLB_{cross} only, and (iii) $LB_{cell}, rLB_{cross}, rLB_{band}$. We vary the trajectory length n and the minimum motif length ξ in Figures 16(a),(b), respectively. The results verify that the bounds complement each other gracefully, and that the performance gains achieved are not due to just one or some of them.

6.2.3 *Effect of Group Size τ*

In GTM (Algorithm 3), the initial group size τ influences the pruning effectiveness and the computation cost of the algorithm. Generally, when τ is small, group-based pruning has a high pruning power but it requires high computation cost. In contrast, when τ is large, group-based pruning becomes faster but it becomes less effective. Figure 17 plots the response time of GTM for different values of τ (x-axis) and trajectory length n (as indicated by the label of each line). We observe that the response time is not overly sensitive to τ. In the following experiments, we set $\tau = 32$ by default as it seems to work well in all cases.

6.3 Performance Evaluation

We compare the performance of our solutions (BTM, GTM, and GTM*) with the baseline (BruteDP) on the real datasets (GeoLife, Truck, and Wild-Baboon). Recall that GTM* is the space-efficient version of GTM.

Figure 18 plots the average response time for different trajectory

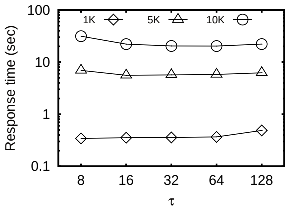
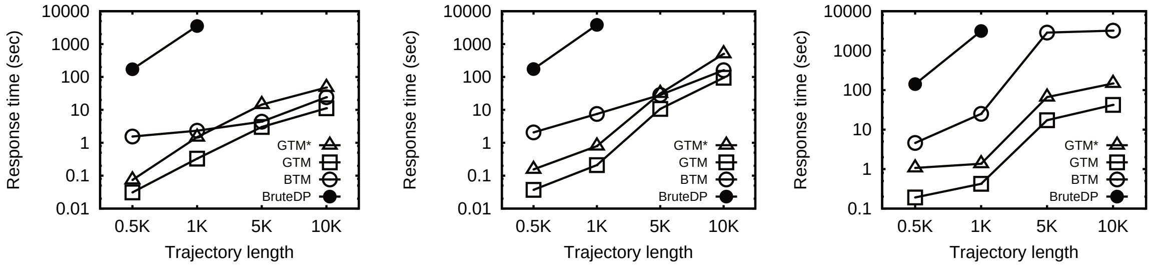
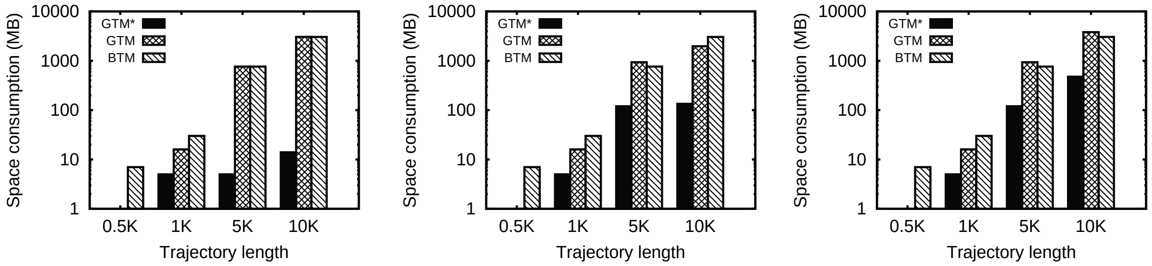


Figure 17: GTM, effect of group size τ



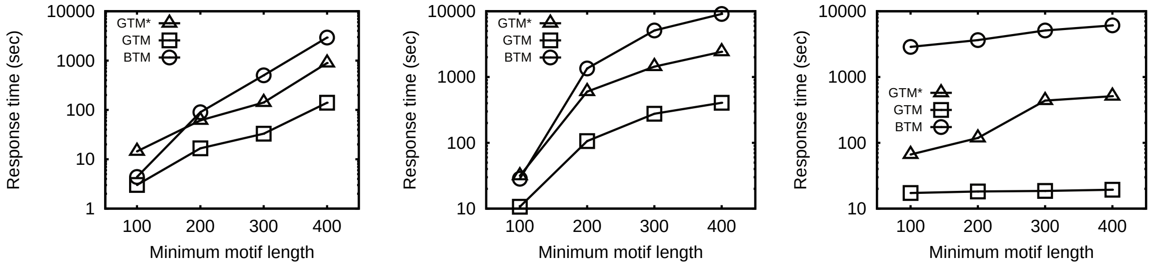
(a) GeoLife (b) Truck (c) Wild-Baboon

Figure 18: Response time vs. trajectory length n



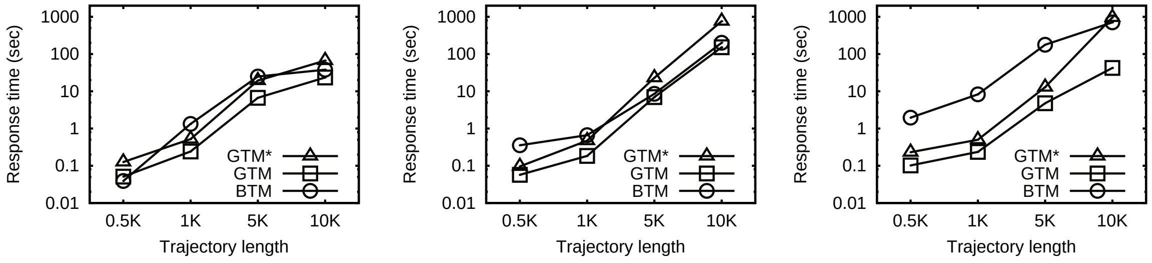
(a) GeoLife (b) Truck (c) Wild-Baboon

Figure 19: Space requirements vs. trajectory length n



(a) GeoLife (b) Truck (c) Wild-Baboon

Figure 20: Response time vs. minimum motif length ξ



(a) GeoLife (b) Truck (c) Wild-Baboon

Figure 21: Response time vs. trajectory length n, two input trajectories

lengths n while fixing $\xi = 100$. BruteDP is prohibitively slow even for small trajectories (e.g., $n = 1000$), thus, we terminate it when it exceeds 2 hours. For the settings where it does terminate within reasonable time, our advanced solutions (i.e., GTM, GTM*) outperform it by 3 orders of magnitude. GTM is the fastest algorithm, with GTM* usually the runner-up. Due to the clear inefficiency of BruteDP, we exclude it from the following experiments.

In Figure 19, we plot the space requirements of BTM, GTM, and GTM* for the same experiment as Figure 18. All methods consume more memory as the trajectory length n increases. As anticipated analytically, the space requirements of BTM and GTM increase sharply with n, but those of GTM* are linear to it. Hence, we consider GTM* as the method of choice for very long trajectories, and the method that strikes the most favourable trade-off between time and space efficiency.

In Figure 20, we measure response time as we vary the minimum trajectory motif length ξ (with n fixed to 5000). The relative performance of the methods is the same as in the previous experiment. The response time of all solutions increases with ξ. That is because a large ξ disqualifies short motifs with small DFDs, thus making it harder to identify early a small bsf that enables aggressive pruning (see also Figure 14(a)).

For completeness, we evaluate our algorithms for motif discovery between different trajectories too. In Figure 21, we randomly select 10 pairs of input trajectories (from the corresponding real dataset) and report the average response time when varying their length n (for fixed $\xi = 100$). The results demonstrate the efficiency of our approaches in this problem variant too. Their performance is very similar to the case of single input trajectory (Problem 1). The same holds when we vary ξ as well as when we measure space requirements; we omit the respective plots to avoid duplication.

7. CONCLUSION

In this paper, we study the trajectory motif discovery problem using the discrete Fréchet distance (DFD). Our contributions include (i) a suite of novel lower bound functions for DFD, (ii) a grouping-based solution that leverages on multi-level pruning to discover the trajectory motif, and (iii) a space-optimized approach that is both time and space efficient. Our fastest solution is over 3 orders of magnitude faster than the baseline solution. All our algorithms are exact. A promising direction for future work is to devise approximate solutions that trade exactness for shorter running times. Another challenging direction is to apply similar optimizations in order to accelerate other trajectory analysis operations that rely on DFD, such as similarity join, subtrajectory clustering, etc.

Acknowledgement

Man Lung Yiu and Bo Tang were supported by grant GRF 152043/15E from the Hong Kong RGC. Kyriakos Mouratidis was supported by the National Research Foundation, Prime Minister's Office, Singapore under its International Research Centres in Singapore Funding Initiative.

8. REFERENCES

[1] P. K. Agarwal, R. B. Avraham, H. Kaplan, and M. Sharir. Computing the discrete fréchet distance in subquadratic time. *SIAM Journal on Computing*, 43(2), 2014.

[2] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On map-matching vehicle tracking data. In *VLDB*, 2005.

[3] K. Buchin, M. Buchin, J. Gudmundsson, M. Löffler, and J. Luo. Detecting commuting patterns by clustering subtrajectories. *IJCGA*, 21(03), 2011.

[4] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing large trajectory data sets with SETI. In *CIDR*, 2003.

[5] E. W. Chambers and Y. Wang. Measuring similarity between curves on 2-manifolds via homotopy area. In *Annual Symposium on Computational Geometry*, 2013.

[6] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, 2005.

[7] S. Dodge. *Exploring movement using similarity analysis*. PhD thesis, 2011.

[8] T. Eiter and H. Mannila. Computing discrete fréchet distance. Technical report, Information Systems Department, Technical University of Vienna, 1994.

[9] E. Frentzos, K. Gratsias, and Y. Theodoridis. Index-based most similar trajectory search. In *ICDE*, 2007.

[10] J. Gudmundsson, P. Laube, and T. Wolle. Computational movement analysis. In *Springer handbook of geographic information*. 2011.

[11] J. Gudmundsson, A. Thom, and J. Vahrenhold. Of motifs and goals: mining trajectory data. In *GIS*, 2012.

[12] J. Gudmundsson and N. Valladares. A gpu approach to subtrajectory clustering using the fréchet distance. *IEEE Transactions on Parallel and Distributed Systems*, 26(4), 2015.

[13] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen. Discovery of convoys in trajectory databases. *PVLDB*, 1(1), 2008.

[14] C. Jia, M. B. Carson, and J. Yu. A fast weak motif-finding algorithm based on community detection in graphs. *BMC bioinformatics*, 14(1), 2013.

[15] M.-P. Kwan. Interactive geovisualization of activity-travel patterns using three-dimensional geographical information systems: a methodological exploration with a large data set. *Transportation Research Part C: Emerging Technologies*, 8(1), 2000.

[16] J.-G. Lee, J. Han, and K.-Y. Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD*, 2007.

[17] X. Li, J. Han, S. Kim, and H. Gonzalez. Roam: Rule-and motif-based anomaly detection in massive moving object data sets. In *SDM*, volume 7, 2007.

[18] Y. Li, U. Leong Hou, M. L. Yiu, and Z. Gong. Quick-motif: An efficient and scalable framework for exact motif discovery. ICDE, 2015.

[19] A. Mueen, E. J. Keogh, Q. Zhu, S. Cash, and M. B. Westover. Exact discovery of time series motifs. In *SDM*, 2009.

[20] T. Oates, A. P. Boedihardjo, J. Lin, C. Chen, S. Frankenstein, and S. Gandhi. Motif discovery in spatial trajectories using grammar inference. In *CIKM*, 2013.

[21] R. W. Sinnott. Virtues of the haversine. *Sky and Telescope*, 68(2), 1984.

[22] R. Sriraghavendra, K. Karthik, and C. Bhattacharyya. Fréchet distance based approach for searching online handwritten documents. In *ICDAR*, volume 1. IEEE, 2007.

[23] A. Strandburg-Peshkin, D. R. Farine, I. D. Couzin, and M. C. Crofoot. Shared decision-making drives collective movement in wild baboons. *Science*, 348(6241), 2015.

[24] K. Toohey and M. Duckham. Trajectory similarity measures. *SIGSPATIAL Special*, 7(1), 2015.

[25] G. Trajcevski, H. Ding, P. Scheuermann, R. Tamassia, and D. Vaccaro. Dynamics-aware similarity of moving objects trajectories. In *GIS*. ACM, 2007.

[26] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering similar multidimensional trajectories. In *ICDE*, 2002.

[27] T. Wylie and B. Zhu. Protein chain pair simplification under the discrete fréchet distance. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 10(6), 2013.

[28] B.-K. Yi, H. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE*, 1998.

[29] Y. Yu, L. Cao, E. A. Rundensteiner, and Q. Wang. Detecting moving object outliers in massive-scale trajectory streams. In *SIGKDD*, 2014.

[30] F. Zambelli, G. Pesole, and G. Pavesi. Motif discovery and transcription factor binding sites before and after the next-generation sequencing era. *Briefings in bioinformatics*, 2012.

[31] Y. Zheng. Trajectory data mining: an overview. *TIST*, 6(3), 2015.

[32] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining interesting locations and travel sequences from gps trajectories. In *WWW*, 2009.

Scheduling Multiple Trips for a Group in Spatial Databases

Roksana Jahan[1], Tanzima Hashem[2], Sukarna Barua[3]
[1,2,3]Department of CSE, Bangladesh University of Engineering and Technology, Bangladesh
{[1]munia_064@yahoo.com} {[2]tanzimahasem, [3]sukarnabarua@cse.buet.ac.bd}

ABSTRACT

Planning user trips in an effective and efficient manner has become an important topic in recent years. In this paper, we introduce Group Trip Scheduling (GTS) queries, a novel query type in spatial databases. Family members normally have many outdoor tasks to perform within a short time for the proper management of home. For example, the members of a family may need to go to a bank to withdraw or deposit money, a pharmacy to buy medicine, or a supermarket to buy groceries. Similarly, organizers of an event may need to visit different points of interests (POIs) such as restaurants and shopping centers to perform many tasks. Given source and destination locations of group members, a GTS query enables a group of n members to schedule n individual trips such that n trips together visit required types of POIs and the total trip distance of n group members is minimized. The trip distance of a group member is measured as the distance between her source to destination via the POIs. We develop an efficient approach to process GTS queries for both Euclidean space and road networks. The number of possible combinations of trips among group members increases with the increase of the number of POIs that in turn increases the query processing overhead. We exploit geometric properties to refine the POI search space and prune POIs to reduce the number of possible combinations of trips among group members. We propose a dynamic programming technique to eliminate the trip combinations that cannot be part of the query answer. We perform experiments using real and synthetic datasets and show that our approach outperforms a straightforward approach with a large margin.

1 Introduction

Family members normally have many outdoor tasks to perform within a short time for the proper management of their home. The members of a family may need to go to a bank to withdraw or deposit money, a pharmacy to buy medicine, or a supermarket to buy groceries. Similarly, organizers of an event may need to visit different points of interests (POIs) like supermarkets, banks, and restaurants to perform many tasks. In reality, all family or organizing members do not need to visit every POI and they can distribute the tasks among themselves. These scenarios motivate us to introduce a *group trip scheduling (GTS) query* that enables a group (e.g., a family) to schedule multiple trips among group members with the minimum total travel distance.

Users have some routine work like traveling from home to office or office to home, and they would prefer to visit other POIs on the way to office or returning home. Given source and destination locations of n group members, a GTS query returns n individual trips such that n trips together visit required types of POIs, each POI type is visited by a single member of the group, and the total trip distance of n group members is minimized. The trip distance of a group member is measured as the distance between her source to destination via the POIs that the group member visits. If the total travel distance is reduced, it will obviously cut down the cost for arranging an event or managing a set of tasks, which is very much desired. In this paper, we propose an efficient approach to process GTS queries for both Euclidean space and road networks.

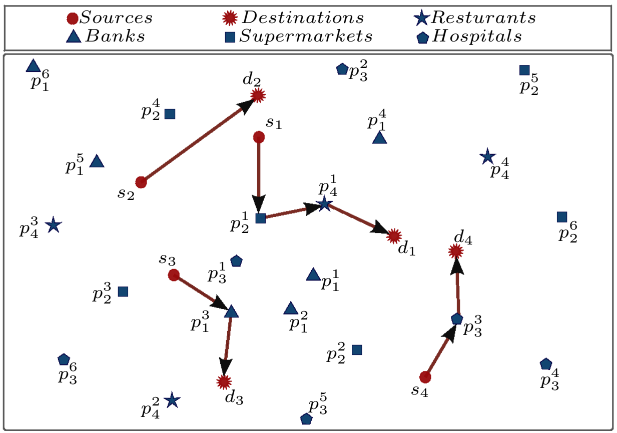


Figure 1: An example of a GTS query

In Figure 1, we consider a group or a family of four members. Every member has preplanned source and destination locations which may be home, office or any other place. Group members u_1, u_2, u_3, and u_4 have source destination pairs, $< s_1, d_1 >$, $< s_2, d_2 >$, $< s_3, d_3 >$, and $< s_4, d_4 >$, respectively. Here, p_j^k denotes a POI of type c_j with ID k. For example, POI p_1^2 in the figure is of type c_1, which represents a bank. The group has to visit four POI types: a bank (c_1), a supermarket (c_2), a hospital (c_3), and a restau-

Series ISSN: 2367-2005

10.5441/002/edbt.2017.35

rant (c_4). For each POI type, there are many options. For example, in real life, banks have many branches in different locations. A GTS query considers all options for each type of POIs, and returns four trips for four group members with the minimum total trip distance, where each POI type is included in a single trip. Figure 1 shows four scheduled trips: $s_1 \rightarrow p_2^1 \rightarrow p_4^1 \rightarrow d_1$, $s_2 \rightarrow d_2$, $s_3 \rightarrow p_1^3 \rightarrow d_3$ and $s_4 \rightarrow p_3^3 \rightarrow d_4$.

A major challenge of our problem is to find the set of POIs from a huge amount of candidate POI sets that provide the optimal answer in real time. For example, California City has about 87635 POIs with 63 different POI types [2]. For each POI type, there are on average 1300 POIs. If the required number of POI types is 4 then the number of candidate POI sets for a GTS query is (1300) × (1300) × (1300) × (1300) = $(1300)^4$ = $2.86e^{+12}$, a huge amount of candidate POI sets. We exploit elliptical properties to bound the POI search space, i.e., to prune POIs that cannot be part of the optimal answer. Though elliptical properties have been explored in the literature for processing other types of spatial queries [5, 7, 12, 13, 18] those pruning techniques are not directly applicable for GTS queries.

Furthermore, a GTS query needs to distribute the POIs of required types in a candidate set among group members. The candidate set contains exactly one POI from each of the m required POI types. The number of possible ways to distribute a candidate POI set of m POIs among n group members is n^m. Thus, the efficiency of a GTS query depends on the refinement of the POI search space and the technique to schedule trips among group members. We develop a dynamic programming technique to reduce the number of possible combinations while scheduling trips among group members. The technique eliminates the trip combinations that cannot be part of the optimal query answer.

Planning trips for a single user or a group in an effective and efficient manner has become an important topic in recent years. A trip planning (TP) query [13] for a single user finds the set of POIs of required types that minimize the trip distance with respect to the user's source and destination locations. To evaluate a GTS query, applying a trip planning algorithm for every user independently for all possible combinations of required POI types requires multiple traversal of the database and would be prohibitively expensive. A group trip planning (GTP) query [8] identifies the set of POIs of required types that minimize the total trip distance with respect to the source and destination locations of group members. In a GTP query, each required POI type is visited by all group members. On the other hand, in a GTS query, separate trips are planned for every group member and each required POI type is visited by only a single group member. For the example scenario mentioned in Figure 1, in Figure 2 we show the resultant trips for a GTP query, where the group members visit all required POI types together. A GTS query is also different from traveling salesman problem (TSP) [11] and its variants [4, 6, 15, 20]. The TSP and its variants assume a limited set of POIs and cannot handle a large dataset like a huge amount of POIs stored in a database.

To the best of our knowledge, we propose the first approach for GTS queries. In summary, the contributions of this paper are as follows:

- We introduce a new type of query, the group trip scheduling (GTS) query in spatial databases.

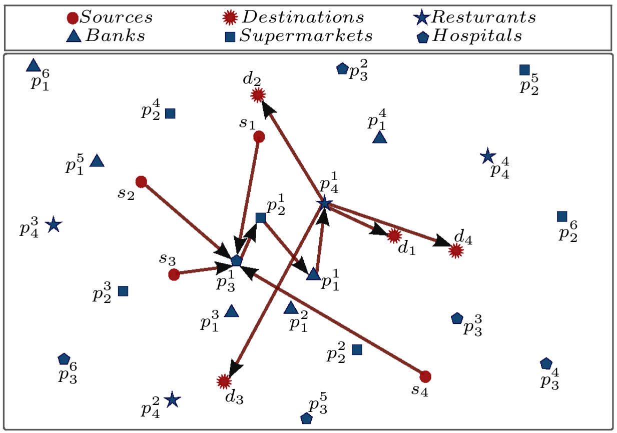


Figure 2: An example of a GTP query

- We present an efficient GTS query processing algorithm. Specifically, we refine the POI search space for processing GTS queries efficiently using elliptical properties and develop an efficient dynamic programming technique to schedule trips among group members.
- We perform extensive experimental evaluation of the proposed techniques and provide an comparative analysis of experimental results using both real and synthetic datasets.

2 Problem Definition

A GTS query for a group is formally defined as follows.

***Definition 1.*[Group Trip Scheduling(GTS) Queries.]** Given a set $\mathbb{P}$ of POIs of different types in a 2-dimensional space, a set of n group members $U = \{u_1, u_2, \ldots, u_n\}$ with independent n source locations $S = \{s_1, s_2, \ldots, s_n\}$ and corresponding n destination locations $D = \{d_1, d_2, \ldots, d_n\}$, and a set of m POI types $\mathbb{C} = \{c_1, c_2, \ldots, c_m\}$, a GTS query returns a set of n trips, $T = \{T_1, T_2, \ldots, T_n\}$ that minimizes the total trip distance, $AggTripDist$ of group members, where a trip T_i corresponds to a group member u_i, group members together visit required types of POIs in $\mathbb{C}$, and a POI type in $\mathbb{C}$ is visited by a single member of the group.

For any two point locations x_1 and x_2 in a 2-dimensional space, let Function $Dist(x_1, x_2)$ return the distance between x_1 and x_2, where the distance can be measured either in the Euclidean space or road networks. The Euclidean distance is measured as the length of the direct line connecting x_1 and x_2. On the other hand, the road network distance is measured as the length of the shortest path between x_1 and x_2 on a given road network graph $\mathbb{G} = (\mathbb{V}, \mathbb{E}, \mathbb{W})$, where each vertex $v \in \mathbb{V}$ represents a road junction, each edge $(v, v') \in \mathbb{E}$ represents a direct path connecting vertices v and v' in $\mathbb{V}$, and each weight $w_{v,v'} \in \mathbb{W}$ represents the length of the direct path represented by the edge (v, v').

A trip T_i of group member u_i starts at s_i, ends at d_i, goes through POIs in A_i, where A_i includes at most m POIs of types specified in $\mathbb{C}$ and $m = |\mathbb{C}| = \sum_{i=1}^{n} |A_i|$. The total trip distance of group members is measured as $AggTripDist = \sum_{i=1}^{n} TripDist_i$. Let p_j denote a POI of type $c_j \in \mathbb{C}$. Without loss of generality, for $A_i = \{p_1, p_2, p_3\}$ and $\{c_1, c_2, c_3\} \in \mathbb{C}$, the trip distance $TripDist_i$ of T_i is computed as $Dist(s_i, p_1) + Dist(p_1, p_2) + Dist(p_2, p_3) + Dist(p_3, d_i)$, if the POI order $p_1 \rightarrow p_2 \rightarrow p_3$ gives the minimum value for $TripDist_i$.

3 System Architecture

Figure 3 shows an overview of the system architecture. The coordinator of the group sends a GTS query request to a location based service provider (LSP). The coordinator provides the source and destination locations of group members and the required POI types that the group members need to visit combinedly. The LSP incrementally retrieves POIs from the database, processes GTS queries and returns scheduled trips to the coordinator of the group that minimizes the total trip distance of the group members.

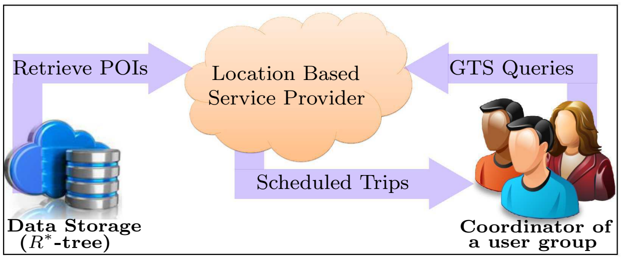

Retrieve POIs
Location Based Service Provider
GTS Queries
Scheduled Trips
Data Storage (R^*-tree)
Coordinator of a user group

Figure 3: System architecture

4 Related Work

Trip planning techniques exist for both single user and group in the literature. Trip planning (TP) queries have been introduced in [12] for a single user. TP queries allow a user to find an optimal route to visit POIs of different types while traveling from her source to destination location. In parallel to the work of TP queries, in [18], Sharifzadeh et al. addressed the optimal sequenced route (OSR) query that also focuses on planning a trip with the minimum travel distance for a single user for a fixed sequence of POI types (e.g., a user first visits a restaurant then a shopping center and a movie theater at the end). In [5], a generalization of the trip planning query, called the multi-rule partial sequenced route (MRPSR) query has been proposed that supports multiple constraints and a partial sequence ordering to visit POI types, and provides a uniform framework to evaluate both of the above mentioned variants [12, 18] of trip planning queries. In [16], the authors proposed an incremental algorithm to find the optimal sequenced route in the Euclidean space and then determine the optimal sequence route in road networks based on the incremental Euclidean restriction. A GTS query is different from TP and OSR queries as GTS queries schedule trips among group members.

A group trip planning query that plans a trip with the minimum aggregate trip distance to visit POIs of different types with respect to source and destination locations of group members has been first proposed in [8]. In [3, 17], the authors proposed efficient algorithms to process GTP queries for a fixed sequence of visiting POI types. In [7], the authors developed an efficient algorithm to process GTP queries in both Euclidean space and road networks. In a GTP query, all group members visit all POI types in their trips, whereas in a GTS query, each POI type is visited by a single member in the group.

A traveling salesman problem (TSP) and variants that focus on planning routes with a limited set of locations are well studied problems in the literature. A generalized traveling salesman problem (GTSP) [6] and multiple traveling salesman problem (MTSP) [4] are well known variations of TSP. A GTSP assumes that from groups of given locations, a salesman visits a location from every group such that the travel distance for the route becomes the minimum. The MTSP allows more than one salesman to be involved in the solution. In MTSP, if the salesmen are initially based at different depots then this variation is known as the multiple depot multiple traveling salesman problem (MDMTSP). However, the limitation of the proposed solutions for TSP and its variants is that they cannot handle a large dataset (e.g., POI data) stored in the database, a scenario that is addressed by a GTS query.

Elliptical properties have been used in the literature to refine the search region for queries like group nearest neighbor queries [14], trip planning queries [12], group trip planning queries [7] and privacy preserving trip planning queries [19]. Though all of these refinement techniques present the refined search region with an ellipse, they differ on the way to set the foci and the length of the major axis of the ellipse. In this paper, we develop two novel techniques to refine the search region using ellipses for GTS queries.

5 Our Approach

In this section, we present our approach to process GTS queries in the Euclidean space and road networks. In a GTS query, the coordinator of a group sends the query request to the LSP and provides required information like group members' source and destination locations, and the required POI types. POI information is indexed using an R^*-tree [1] in the database. The LSP incrementally retrieves POIs from the database until it identifies the trips that minimize the total travel distance of the group members. The underlying idea of the efficiency of our approach is the POI search region refinement techniques using elliptical properties and the dynamic programming technique to schedule multiple trips among the group members.

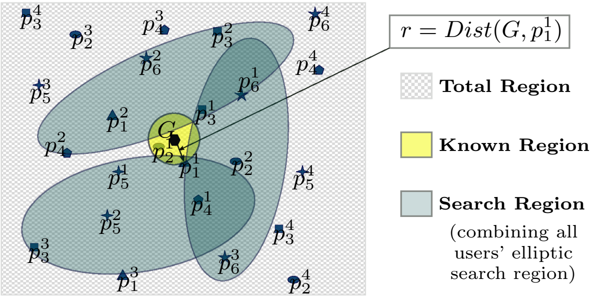


Figure 4: Known region and search region

We use the concept of known region and search region [7, 12] for the retrieval of POIs from the database. The known region represents the area which has already been explored, that means all POIs inside the known region have been retrieved from the database. The search region represents the refined space that we need to explore for the optimal solution. In Figure 4, suppose the LSP retrieves the nearest POIs p_2^1 and p_1^1 with respect to the geometric centroid G of source and destination locations of a group of three members, where p_1^1 is the farthest POI from G among POIs p_2^1 and p_1^1 that have been already retrieved. The circular region centered at G with radius equal to the distance between G and p_1^1 is the known region. We refine the POI search region with respect to the retrieved POIs in the known region using multiple ellipses, and call it simply a search region. In

Figure 4, based on current retrieved POIs, p_2^1 and p_1^1, the search region is the union of three ellipses.

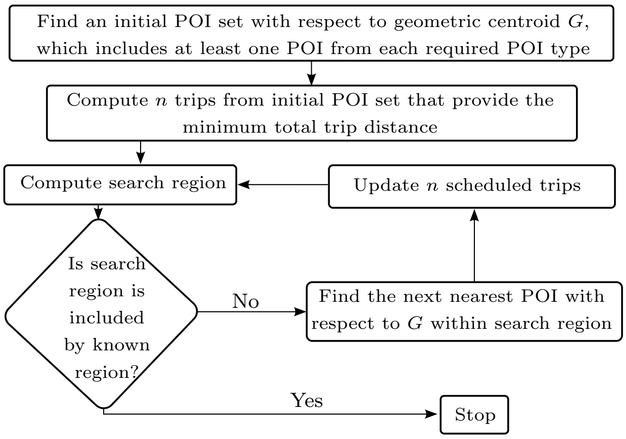

Find an initial POI set with respect to geometric centroid G, which includes at least one POI from each required POI type
Compute n trips from initial POI set that provide the minimum total trip distance
Compute search region
Update n scheduled trips
Is search region is included by known region?
No
Find the next nearest POI with respect to G within search region
Yes
Stop

Figure 5: Overview of our approach for GTS queries

Figure 5 shows an overview of our developed approach for processing GTS queries. Our approach initially incrementally retrieves the nearest POIs from G until at least one POI from each required POI type has been retrieved. Using the initial retrieved POI set, our approach schedules n trips that provide the minimum total travel distance for the group members, and refines the search region to prune POIs that cannot be the part of the query answer. Then the proposed approach checks whether the known region includes the search region. If yes, then our approach has retrieved all POIs that are required to find the optimal answer and the approach terminates the search. Otherwise, our approach continues to incrementally retrieve the next nearest POIs within the search region, updates scheduled n trips, refines the search region, and checks the termination condition of the search until the condition becomes true. In the following sections, we elaborate the steps of our approach for processing GTS queries.

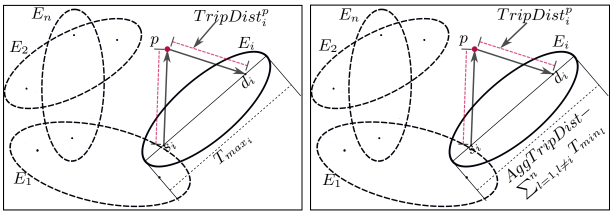
5.1 Computing the known region

For both Euclidean and road network spaces, our approach incrementally retrieves the Euclidean nearest POIs with respect to the geometric centroid G of n source-destination pairs of group members. It uses the best-first search (BFS) to find the POIs of required POI types that are assumed to be indexed using an R^*-tree [1] in the database. The BFS search also prunes the POIs whose types do not match with the required POI types and returns the remaining POIs.

Let the BFS discover p_j as the first nearest POI with respect to G. The circular region centered at G with radius r equal to the Euclidean distance between G and p_j is the known region. With the retrieval of the next nearest POI, r is updated with the Euclidean distance from G to the last retrieved nearest POI from the database.

5.2 Refinement of the search region

The key idea of our search region refinement techniques is based on elliptical properties. A smaller search region decreases the number of POIs retrieved from the database, avoids unnecessary trip computations, and reduces I/O access and computational overhead significantly. We present two novel techniques in Theorems 1 and 2 to refine the search region using multiple ellipses, and based on these two refinement techniques, we develop our algorithm to process GTS queries in Section 5.5. The notations that we use in our theorems are summarized below:



(a) Proof of Theorem 1

(b) Proof of Theorem 2

Figure 6: Search region refinement

- T_{min_i}: the minimum trip distance for a group member u_i, i.e., the distance between s_i and d_i without visiting any POI type.
- T_{max_i}: the maximum trip distance for a group member u_i, i.e., the trip distance from s_i to d_i via required m POI types.
- $TripDist_i$: the current trip distance of a group member u_i among the scheduled trips.
- $AggTripDist$: the current minimum total trip distance of the group.

Above notations are measured in terms of Euclidean distances if a GTS query is evaluated in the Euclidean space, and in terms of road network distances if a GTS query is evaluated in the road networks. Theorems 1 and 2 show two ways to refine the search region for a GTS query in the Euclidean space and road networks.

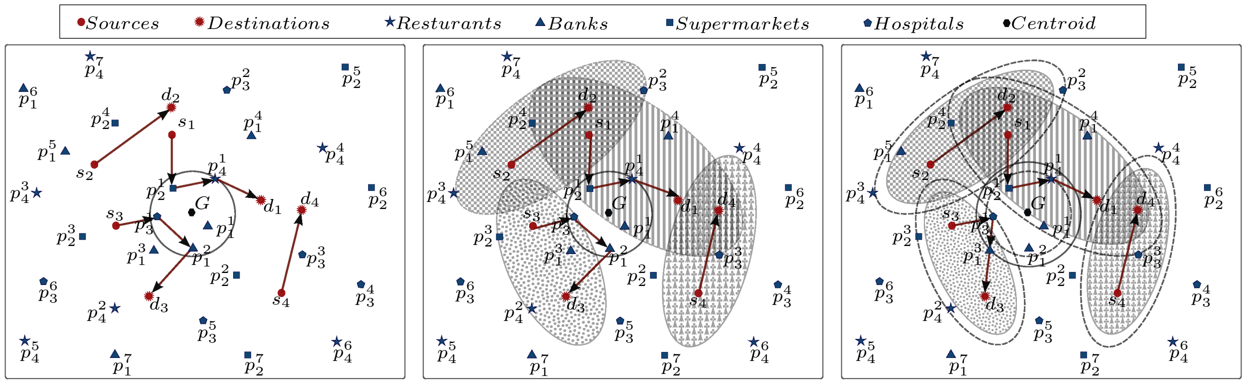
THEOREM 1. *The search region can be refined as the union of n ellipses $E_1 \cup E_2 \cup \ldots \cup E_n$, where the foci of ellipse E_i are at s_i and d_i, and the major axis of the ellipse E_i is equal to T_{max_i}.*

PROOF. Let a POI p lie outside the search region, $E_1 \cup E_2 \cup \ldots \cup E_n$, and $AggTripDist^p$ be the total trip distance of the group, where a group member u_i's trip includes POI p as shown in Figure 6(a). We have to prove that POI p can not be a part of the optimal solution, i.e., $AggTripDist^p > AggTripDist$. Let $TripDist_i^p$ be the trip distance for the group member u_i whose trip includes POI p. An elliptical property states that the Euclidean distance between two foci via a point outside the ellipse is greater than the length of the major axis. Since the road network distance is greater than or equal to the Euclidean distance, the road network distance between two foci via a point outside the ellipse is also greater than the length of the major axis. As POI p lies outside the ellipse E_i, for both Euclidean and road network spaces we have,

$$TripDist_i^p > T_{max_i} \quad (1)$$

T_{max_i} represents the trip distance of user u_i for visiting m POI types. Any trip passing through the POI p outside the ellipse E_i can not give better trip distance for user u_i. Thus, any POI outside the union of ellipses $E_1, E_2, \ldots, E_n$ can not improve the total trip distance $AggTripDist$ for the group and can not be a part of an optimally scheduled group of trips. Thus, $AggTripDist^p > AggTripDist$. □

THEOREM 2. *The search region can be refined as the union of n ellipses $E_1 \cup E_2 \cup \ldots \cup E_n$, where the foci of ellipse E_i are at s_i and d_i, and the major axis of the ellipse is equal to $AggTripDist - \sum_{l=1,l\neq i}^{n} T_{min_l}$.*



(a) Initial known region (the circle with center G) and scheduled trips calculated using initial POIs

(b) Refined search region

(c) Known region expands (outer circle) and search region shrinks (inner ellipses)

Figure 7: Steps of our approach

PROOF. Let a POI p lie outside the search region, $E_1 \cup E_2 \cup \ldots \cup E_n$, and $AggTripDist^p$ be the total trip distance of the group, where a group member u_i's trip includes POI p as shown in Figure 6(b). We have to prove that POI p can not be a part of the optimal solution, i.e., $AggTripDist^p > AggTripDist$.

Let $TripDist_i^p$ be the trip distance for the group member u_i whose trip includes POI p. An elliptical property states that the Euclidean distance between two foci via a point outside the ellipse is greater than the length of the major axis. Since the road network distance is greater than or equal to the Euclidean distance, the road network distance between two foci via a point outside the ellipse is also greater than the length of the major axis. As the POI p lies outside the ellipse E_i, for both Euclidean and road network spaces we have,

$$TripDist_i^p > AggTripDist - \sum_{l=1,l\neq i}^{n} T_{min_l}$$

Rearranging the equation we get,

$$TripDist_i^p + \sum_{l=1,l\neq i}^{n} T_{min_l} > AggTripDist \qquad (2)$$

By definition we know,

$$AggTripDist^p = TripDist_i^p + \sum_{l=1,l\neq i}^{n} TripDist_l^p \qquad (3)$$

and

$$\sum_{l=1,l\neq i}^{n} TripDist_l^p \geq \sum_{l=1,l\neq i}^{n} T_{min_l} \qquad (4)$$

From Equations 3 and 4, we get,

$$AggTripDist^p \geq TripDist_i^p + \sum_{l=1,l\neq i}^{n} T_{min_l} \qquad (5)$$

Combining inequalities of 2 and 5,

$$AggTripDist^p > AggTripDist$$

Thus, any POI outside the search region $E_1 \cup E_2 \cup \ldots \cup E_n$ can not improve the total trip distance for the group and can not be a part of an optimally scheduled group of trips. □

Our approach refines the ellipses of every group member independently using both bounds proposed in Theorems 1 and 2, and selects the bound that provides the minimum length for the major axis of the ellipse. For the same foci, the smaller major axis represents a smaller ellipse. It may happen that for an ellipse of a member, Theorem 1 provides the minimum length of the major axis and for another member's ellipse, Theorem 2 provides the minimum length of the major axis. The refined search region is computed as the union of the smaller ellipses of all group members.

For a GTS query, our approach retrieves an initial set of nearest POIs that includes at least one POI of each required type. From the initial set of POIs, our approach schedules trips with the minimum total trip distance for the group using the dynamic programming technique shown in Section 5.4, and refines the search region using Theorems 1 and 2. With the incremental retrieval of the nearest POIs from G within the refined search region, our approach checks and updates the scheduled trips, if the newly discovered POIs improve the current scheduled trips. The newly updated trips may improve the bound T_{max_i} for a group member or the total trip distance of the group $AggTripDist$, which can further refine the search region.

Figure 7(a) shows the initial set of retrieved POIs $p_1^1, p_1^2, p_2^1, p_3^1, p_4^1$, the known region, and four scheduled trips using the initial POI set for a group of four members. Note that the initial set may include more than one POIs of same POI type (e.g., p_1^1 and p_1^2) because the incremental nearest POI retrieval continues until the initial set includes at least one POI from every required POI type. Using bounds from Theorem 1 and 2, we compute and refine the search region. Figure 7(b) shows the refined search region as the union of four ellipses. After retrieving the next nearest POI p_1^3, the known region expands, which has the radius equal to $Dist(G, p_1^3)$. Our approach checks whether this new POI can improve the current solution. In this example, the new POI p_1^3 decreases the trip distance for group member u_3 and thus, the updated trip for u_3 is $s_3 \to p_3^1 \to p_1^3 \to d_3$. It also improves the total trip distance and shrinks the search region for all group members. In Figure 7(c), the dotted lines show the scenario before retrieving POI p_1^3 and the shaded areas with solid lines show the updated scenario after retrieving the POI p_1^3. With the retrieval of the nearest POIs from the database, the known region expands and the search region shrinks or remains same.

5.3 Terminating condition for POI retrieval

When the known region covers the search region, no more minimization in the total trip distance is further possible. At this point, we can terminate traversing R^*-tree and retrieving POIs. Figure 8 shows that the known region covers the search region.

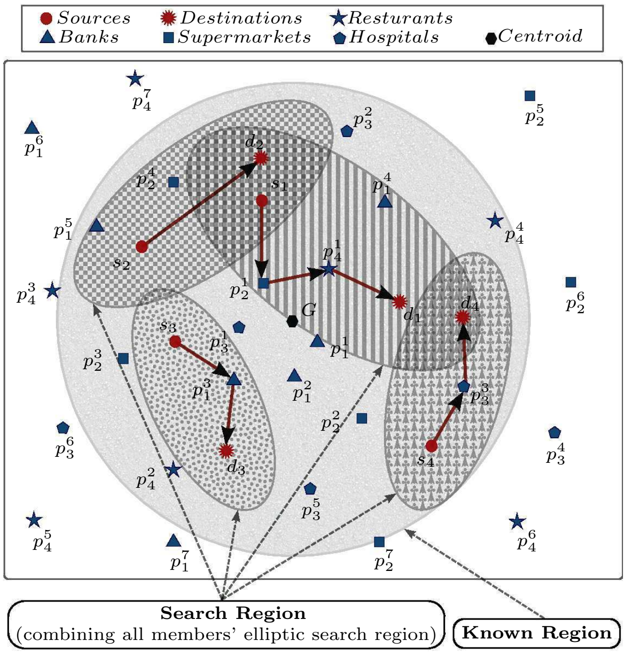


Figure 8: Terminating condition: the known region includes the search region

5.4 Dynamic programming technique for scheduling trips

Scheduling the trips among the group members is an essential component of GTS query processing approach. After retrieving the initial POI set, our approach schedules the trips among the group members such that the total trip distance of the group is minimized. Each time our approach retrieves new POIs, it again schedules trips using new POIs, if the new trips improve the total trip distance of the group. Thus, the efficiency of our approach largely depends on the computational cost of scheduling trips among the group members. We propose a dynamic programming technique to schedule the trips among the group members. The technique reduces the number of trip combinations that we need to consider to find the set of trips with the minimum total trip distance. The distances computed in our dynamic programming technique are Euclidean distances, if a GTS query is processed in the Euclidean space, and the distances are road network distances, otherwise.

Our dynamic programming technique minimizes the following objective function:

$$\sum_{i=1}^{n} TripDist_i$$

satisfying constraints that a group of n members together visit m different POI types and each POI type is visited by a single group member. Let $\mathbb{C}_{T_i}$ be the set of POI types visited by trip T_i of user u_i, where $0 \leq |\mathbb{C}_{T_i}| \leq m$. Formal representation of the constraints are as follows. The dynamic programming technique satisfies,

$$\sum_{i=1}^{n} |\mathbb{C}_{T_i}| = m, \quad \bigcup_{i=1}^{n} \mathbb{C}_{T_i} = \mathbb{C} \text{ and } \forall_{i,j}(\mathbb{C}_{T_i} \cap \mathbb{C}_{T_j}) = \emptyset$$

For the GTS query, we have a set of m POI types

Table 1: Structure of dynamic table ν_y, where $0 \leq y \leq (m-1)$

	$\{u_1\}$	...	$\{u_n\}$	$\{u_1u_2\}$	...	$\{u_1u_2 \dots u_{n-1}\}$
$\{c_1, c_2, \dots, c_y\}$						
$\{c_1, c_3, \dots, c_y\}$						
$\vdots$						

Table 2: Structure of dynamic table ν_m

	$\{u_1\}$	...	$\{u_n\}$	$\{u_1u_2\}$	...	$\{u_1u_2 \dots u_n\}$
$\{c_1, c_2, \dots, c_m\}$						

$\mathbb{C}=\{c_1, c_2, \dots, c_m\}$, where a group member visits any number of POI types from 0 to m. Thus, there are $\sum_{y=0}^{m}({}^{m}C_y)$ ways to choose any y POI types from $m(=|\mathbb{C}|)$ different POI types, where $0 \leq y \leq m$. Suppose ${}^{\mathbb{C}}C_y$ denotes the set of all possible y chooses from the set of POI types $\mathbb{C}$. Let $({}^{\mathbb{C}}C_y)^j$ represent the jth member of the set ${}^{\mathbb{C}}C_y$. Suppose we have a set of $m = |\mathbb{C}| = 4$ POI types, $\mathbb{C} = \{c_1, c_2, c_3, c_4\}$. For $y = 2$, the number of ways to choose y POI types from $m(=|\mathbb{C}|)$ POI types is ${}^{|\mathbb{C}|}C_y = {}^{4}C_2 = 6$ and the set all possible y chooses from the set $\mathbb{C}$ is ${}^{\mathbb{C}}C_y = \{\{c_1, c_2\}, \{c_1, c_3\}, \{c_1, c_4\}, \{c_2, c_3\}, \{c_2, c_4\}, \{c_3, c_4\}\}$, where $({}^{\mathbb{C}}C_y)^1 = \{c_1, c_2\}$, $({}^{\mathbb{C}}C_y)^2 = \{c_1, c_3\}, \dots, ({}^{\mathbb{C}}C_y)^6 = \{c_3, c_4\}$.

For each member of the set ${}^{\mathbb{C}}C_y$, we calculate optimal trips for each group member in $U = \{u_1, u_2, u_3, \dots, u_n\}$ and store trip distances for future computations. This is the initial step for our dynamic programming technique. We define $m+1$ dynamic tables, $\nu_0, \nu_1, \nu_2, \dots \nu_m$ to store the trip distances of every group member and the combined trip distances of the group members. Table ν_y has ${}^{m}C_y$ rows, where jth row corresponds to jth member of the set ${}^{\mathbb{C}}C_y$, i.e., $({}^{\mathbb{C}}C_y)^j$.

Each table has two types of columns : ***single member columns*** and ***combined member columns***. Each table has n single member columns, where each column corresponds to a member of the group $U = \{u_1, u_2, u_3, \dots, u_n\}$. The cells of these columns store the minimum trip distances for the corresponding column's member to visit the POI types of the corresponding rows. Each dynamic table except ν_m has $(n-2)$ combined member columns $u_1u_2, u_1u_2u_3, \dots, u_1u_2..u_{n-1}$, where the cells of the corresponding columns store the combined trip distances of the corresponding column's multiple members. For example, each cell of the column u_1u_2 stores the minimum combined trip distance of user u_1 and u_2 to visit the POI types of the corresponding row, where a POI type is visited either by u_1 or u_2. Table 1 shows the structure of ν_y where $0 \leq y \leq (m-1)$. Table 2 shows the structure of ν_m that has an extra column $u_1u_2 \dots u_n$ to store the minimum total trip distance for n scheduled trips, where n trips together visit m required POI types and every POI type is visited by a single trip. The table has only one row which contains all m POI types.

In addition to storing the minimum trip distance, each cell of the dynamic tables stores the set of POIs for which

Table 3: Possible number of POI type distributions between u_1 and u_2

u_1	u_2
3	0
2	1
1	2
0	3

Table 4: Candidate trips with trip distances for cell $\nu_2[\{c_1, c_2\}][\{u_1\}]$

Candidate trips	Distances
$s_1 \to p_2^1 \to p_1^1 \to d_1$	65.55
$s_1 \to p_2^1 \to p_1^2 \to d_1$	61.72
$s_1 \to p_1^1 \to p_2^1 \to d_1$	60.44
$s_1 \to p_1^2 \to p_2^1 \to d_1$	51.58

Table 5: Dynamic tables for the example scenario

(a) Dynamic table ν_0

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\emptyset$	51.55	93.33	68.84	81.78	144.88	213.72

(b) Dynamic table ν_1

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1\}$	51.57	96.22	123.61	90.67	144.90	213.74
$\{c_2\}$	51.56	93.33	68.84	81.78	144.88	213.72
$\{c_3\}$	51.55	93.97	78.31	81.79	144.88	213.72
$\{c_4\}$	51.55	93.33	68.84	81.78	144.88	213.72

(c) Dynamic table ν_2

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1, c_2\}$	51.58	96.22	123.61	90.67	144.90	213.74
$\{c_1, c_3\}$	51.86	96.26	123.68	90.70	145.19	214.03
$\{c_1, c_4\}$	51.57	96.23	123.61	90.67	144.90	213.74
$\{c_2, c_3\}$	51.57	93.97	78.34	81.81	144.88	213.72
$\{c_2, c_4\}$	51.56	93.34	68.84	81.78	144.88	213.72
$\{c_3, c_4\}$	51.55	93.97	78.32	81.79	144.88	213.72

(d) Dynamic table ν_3

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$
$\{c_1, c_2, c_3\}$	51.90	96.26	123.68	90.70	145.19	214.03
$\{c_1, c_2, c_4\}$	51.59	96.23	123.61	90.67	144.90	213.74
$\{c_1, c_3, c_4\}$	51.88	96.28	123.68	90.71	145.19	214.03
$\{c_2, c_3, c_4\}$	51.57	93.97	78.34	81.81	144.88	213.72

(e) Dynamic table ν_4

	$\{u_1\}$	$\{u_2\}$	$\{u_3\}$	$\{u_4\}$	$\{u_1u_2\}$	$\{u_1u_2u_3\}$	$\{u_1u_2u_3u_4\}$
$\{c_1, c_2, c_3, c_4\}$	51.90	96.28	123.69	90.71	145.20	214.03	295.53

the minimum trip distance is obtained. For example, cell $\nu_3[\{c_1, c_3, c_4\}][\{u_1\}]$ stores the minimum trip distance and the POI set $< p_3, p_1, p_4 >$, for which u_1 obtains the minimum trip distance.

The size of a dynamic table ν_y is : ${}^mC_y \times (n + (n - 2))$, where $0 \leq y \leq (m-1)$, and the size of table ν_m is ${}^mC_m \times (n + (n-2)+1)$. Thus, the total space required for dynamic tables is $\sum_{y=0}^{(m-1)}({}^mC_y \times (n+(n-2))) + ({}^mC_m \times (n+(n-2)+1)) = (2^{m+1} \times (n-1)+1)$ units. Similarly, the processing time of the dynamic programming technique is proportional to the number of the dynamic tables and the number of cells in a dynamic table, which vary with the values of m and n.

Contents of cells of the single member columns of a dynamic table are computed using already retrieved POIs from the database. To compute the contents of cells of the combined member columns of a dynamic table ν_y, we use the single member columns of the same table, and both single and combined member columns of $\nu_0, \nu_1, \ldots, \nu_{y-1}$. For example, for computing each cell of combined member column u_1u_2 of ν_4, we use the already calculated single member columns of ν_4, and both single and combined member columns of ν_0, ν_1, ν_2 and ν_3 based on possible number of POI type distributions between members u_1 and u_2 of that corresponding column. For the example scenario, to visit 3 POI types, possible ways to distribute the number of POI types between u_1 and u_2 are listed in Table 3. Formally, the minimum trip distance stored in a cell (e.g., $\nu_y[\{c_1, c_2, \ldots, c_y\}][\{u_1u_2\}]$ of table ν_y) is computed as $\min_{g=0}^{y}\{\min_{j=1}^{{}^mC_g}\{\min_{k=1}^{{}^mC_{y-g}}(\nu_g[({}^{\mathbb{C}}C_g)^j][\{u_1\}] + \nu_{(y-g)}[({}^{\mathbb{C}}C_{(y-g)})^k][\{u_2\}])\}\}$, where $({}^{\mathbb{C}}C_g)^j \cap ({}^{\mathbb{C}}C_{(y-g)})^k = \emptyset$. The constraint guarantees that no POI type is considered twice while computing the minimum trip distance.

Similar to the combined member column u_1u_2, for computing each cell of combined member column $u_1u_2u_3$ of ν_4, we use the same dynamic tables, and similar distribution listed in Table 3 between combined members u_1u_2 (instead of u_1) and single member u_3 (instead of u_2). Thus, we incrementally compute dynamic tables $\nu_0, \nu_1, \nu_2, \ldots, \nu_m$, one by one and finally we get our desired result for a GTS query.

We elaborate our dynamic programming technique with an example. Suppose a group of 4 members, $\{u_1, u_2, u_3, u_4\}$, together want to visit 4 POI types $\{c_1, c_2, c_3, c_4\}$ with the minimum total trip distance, and each POI type is visited by a single member. Here, $n = 4$, $m = 4$, and a group member can visit any number of POI types between 0 to m.

Figure 7(a) shows the initial set of retrieved POIs: $p_1^1, p_1^2, p_2^1, p_3^1, p_4^1$ and the known region. The initial set includes at least a POI from every POI type. Using these POIs, we first compute all possible trips for the group members and then schedule the trips using our proposed dynamic programming technique.

We define $(m + 1)$, i.e., 5 tables, ν_0, ν_1, ν_2, ν_3 and ν_4 to store the computed trip distances and combined trip distances of the group members. Each dynamic table ν_y has ${}^{m=4}C_y$ rows, where each row corresponds to a member of the set ${}^{\mathbb{C}}C_y$. Each table has $n = 4$ single member columns, where a column corresponds to a group member in $\{u_1, u_2, u_3, u_4\}$, and $n - 2 = 2$ combined member columns, u_1u_2 and $u_1u_2u_3$. Table ν_4 contains an extra column $u_1u_2u_3u_4$ to store the minimum total trip distance of the 4 scheduled trips for 4 users that together visit 4 POI types, where each POI type is visited by a single user. Tables 5 (a-e) show ν_0, ν_1, ν_2, ν_3 and ν_4 for the considered example.

Computing single member columns: In the dynamic tables, columns u_1, u_2, u_3 and u_4 are the single member columns. Each cell of these columns of a table stores the minimum trip distance for the corresponding column's user passing through POI types of the corresponding row of that table. For example, in Table 5(c), cell $\nu_2[\{c_1, c_2\}][\{u_1\}]$ contains the minimum trip distance for user u_1 passing through POI types c_1 and c_2. For computing this trip distance, we consider user, u_1's source (s_1) and destination (d_1) locations along with candidate POIs in the initial set: $\{p_1^1, p_1^2\}$ and $\{p_2^1\}$ with POI types c_1 and c_2, respectively. All candidate trips for cell $\nu_2[\{c_1, c_2\}][\{u_1\}]$ using these POIs with the corresponding trip distances are listed in Table 4.

Among the candidate trips listed in this table, the minimum trip distance 51.58 for trip $s_1 \rightarrow p_1^2 \rightarrow p_2^1 \rightarrow d_1$ is stored in cell $\nu_2[\{c_1, c_2\}][\{u_1\}]$. Similarly, our dynamic programming technique populates all cells of the single member columns of ν_1, ν_2, ν_3 and ν_4. Table ν_0 is a trivial one that stores trip distances for particular user's trip from her source to destination location only.

Computing combined member columns: Using the single member columns and already calculated combined member columns, we dynamically calculate the combined member columns of ν_0, ν_1, ν_2, ν_3 and ν_4 one by one.

In ν_0, cell $\nu_0[\emptyset][\{u_1u_2\}]$ contains the minimum total trip distance of trips T_1 and T_2, where the trips correspond to users u_1 and u_2, respectively, and visit no POI type. Table 6 shows the candidate combinations that are used to compute the cell value, where trip distances are for users' trips from their source to destination locations.

Table 6: Candidate combined combinations with trip distances for cell $\nu_0[\emptyset][\{u_1u_2\}]$

Combined Combinations	Distances	Total
$\nu_0[\emptyset][\{u_1\}]$ + $\nu_0[\emptyset][\{u_2\}]$	51.55 + 93.33	144.88

Table 7: Candidate combined combinations with trip distances for cell $\nu_1[\{c_1\}][\{u_1u_2\}]$

Combined Combinations	Distances	Total
$\nu_1[\{c_1\}][\{u_1\}]$ + $\nu_0[\emptyset][\{u_2\}]$	51.57 + 93.33	144.90
$\nu_0[\emptyset][\{u_1\}]$ + $\nu_1[\{c_1\}][\{u_2\}]$	51.55 + 96.22	147.77

To compute the cells of the combined member columns for other table ν_y, we need to consider all dynamic tables from ν_0 to ν_y. For example, in ν_2, cell $\nu_2[\{c_1, c_2\}][\{u_1u_2\}]$ stores the minimum total trip distance of trips T_1 and T_2, where the trips correspond to users u_1 and u_2, respectively. Here a user (u_1 or u_2) can visit any number (0 or 1 or 2) of POI types, but u_1 and u_2 together visit the POI types $\{c_1, c_2\}$, and each POI type is either visited by u_1 or u_2. For computing the cell value, we use stored single member trip distances and multiple member trip distances in ν_0, ν_1 and ν_2. Using ν_0, ν_1 and ν_2 (Tables 5(a-c)), Table 8 shows the candidate combinations of POI types for u_1 and u_2 along with the trip distances for computing the value for cell $\nu_2[\{c_1, c_2\}][\{u_1u_2\}]$ in ν_2 (Table 5(c)). Among candidate combinations listed in Table 8, the minimum total trip distance 144.90 is stored in cell $\nu_2[\{c_1, c_2\}][\{u_1u_2\}]$.

Table 8: Candidate combined combinations with trip distances for cell $\nu_2[\{c_1, c_2\}][\{u_1u_2\}]$

Combined Combinations	Distances	Total
$\nu_2[\{c_1, c_2\}][\{u_1\}]$ + $\nu_0[\emptyset][\{u_2\}]$	51.58 + 93.33	144.91
$\nu_1[\{c_1\}][\{u_1\}]$ + $\nu_1[\{c_2\}][\{u_2\}]$	51.57 + 93.33	144.90
$\nu_1[\{c_2\}][\{u_1\}]$ + $\nu_1[\{c_1\}][\{u_2\}]$	51.56 + 96.22	147.78
$\nu_0[\emptyset][\{u_1\}]$ + $\nu_2[\{c_1, c_2\}][\{u_2\}]$	51.55 + 96.22	147.77

Similarly, our dynamic programming technique populates all cells of the combined member columns of ν_0, ν_1, ν_2, ν_3 and ν_4. Candidate combinations with trip distances for cell $\nu_1[\{c_1\}][\{u_1u_2\}]$, $\nu_3[\{c_1, c_2, c_3\}][\{u_1u_2\}]$ and $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1u_2\}]$ are listed in Table 7, Table 9 and Table 10, respectively.

Table 9: Candidate combined combinations with trip distances for cell $\nu_3[\{c_1, c_2, c_3\}][\{u_1u_2\}]$

Combined Combinations	Distances	Total
$\nu_3[\{c_1, c_2, c_3\}][\{u_1\}]$ + $\nu_0[\emptyset][\{u_2\}]$	51.90 + 93.33	145.23
$\nu_2[\{c_1, c_2\}][\{u_1\}]$ + $\nu_1[\{c_3\}][\{u_2\}]$	51.58 + 93.97	145.55
$\nu_2[\{c_1, c_3\}][\{u_1\}]$ + $\nu_1[\{c_2\}][\{u_2\}]$	51.86 + 93.33	145.19
$\nu_2[\{c_2, c_3\}][\{u_1\}]$ + $\nu_1[\{c_1\}][\{u_2\}]$	51.57 + 96.22	147.79
$\nu_1[\{c_1\}][\{u_1\}]$ + $\nu_2[\{c_2, c_3\}][\{u_2\}]$	51.55 + 96.22	147.77
$\nu_1[\{c_2\}][\{u_1\}]$ + $\nu_2[\{c_1, c_3\}][\{u_2\}]$	51.56 + 96.26	147.82
$\nu_1[\{c_3\}][\{u_1\}]$ + $\nu_2[\{c_1, c_2\}][\{u_2\}]$	51.57 + 93.97	145.54
$\nu_0[\emptyset][\{u_1\}]$ + $\nu_3[\{c_1, c_2, c_3\}][\{u_2\}]$	51.55 + 96.26	147.81

Table 10: Candidate combined combinations with trip distances for cell $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1u_2\}]$

Combined Combinations	Distances	Total
$\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1\}]$+$\nu_0[\emptyset][\{u_2\}]$	51.90+93.33	145.23
$\nu_3[\{c_1, c_2, c_3\}][\{u_1\}]$+$\nu_1[\{c_4\}][\{u_2\}]$	51.90+93.33	145.23
$\nu_3[\{c_1, c_2, c_4\}][\{u_1\}]$+$\nu_1[\{c_3\}][\{u_2\}]$	51.59+93.97	145.56
$\nu_3[\{c_1, c_3, c_4\}][\{u_1\}]$+$\nu_1[\{c_2\}][\{u_2\}]$	51.88+93.33	145.21
$\nu_3[\{c_2, c_3, c_4\}][\{u_1\}]$+$\nu_1[\{c_1\}][\{u_2\}]$	51.57+96.22	147.79
$\nu_2[\{c_1, c_2\}][\{u_1\}]$+$\nu_2[\{c_3, c_4\}][\{u_2\}]$	51.58+93.97	145.55
$\nu_2[\{c_1, c_3\}][\{u_1\}]$+$\nu_2[\{c_2, c_4\}][\{u_2\}]$	51.86+93.34	145.20
$\nu_2[\{c_1, c_4\}][\{u_1\}]$+$\nu_2[\{c_2, c_3\}][\{u_2\}]$	51.57+93.97	145.54
$\nu_2[\{c_2, c_3\}][\{u_1\}]$+$\nu_2[\{c_1, c_3\}][\{u_2\}]$	51.57+96.23	147.80
$\nu_2[\{c_2, c_4\}][\{u_1\}]$+$\nu_2[\{c_1, c_3\}][\{u_2\}]$	51.56+96.26	147.82
$\nu_2[\{c_3, c_4\}][\{u_1\}]$+$\nu_2[\{c_1, c_2\}][\{u_2\}]$	51.55+96.22	147.77
$\nu_1[\{c_1\}][\{u_1\}]$+$\nu_3[\{c_2, c_3, c_4\}][\{u_2\}]$	51.55+96.26	147.81
$\nu_1[\{c_2\}][\{u_1\}]$+$\nu_3[\{c_1, c_3, c_4\}][\{u_2\}]$	51.55+96.23	147.78
$\nu_1[\{c_3\}][\{u_1\}]$+$\nu_3[\{c_1, c_2, c_4\}][\{u_2\}]$	51.56+96.28	147.84
$\nu_1[\{c_4\}][\{u_1\}]$+$\nu_3[\{c_1, c_2, c_3\}][\{u_2\}]$	51.57+93.97	145.54
$\nu_0[\emptyset][\{u_1\}]$+$\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_2\}]$	51.55+96.28	147.83

We gradually combine trips of other users, u_3 and u_4, and update the other combined member columns one by one. For example, in ν_2, cell $\nu_2[\{c_1, c_2\}][\{u_1u_2u_3\}]$ contains the minimum total trip distance of trips T_1, T_2 and T_3, where the trips correspond to users u_1, u_2 and u_3, respectively, and together visit the POI types $\{c_1, c_2\}$. Using ν_0, ν_1 and ν_2 (Tables 5(a-c)), Table 11 shows the candidate combinations of POI types for combined members u_1u_2 and single member u_3 along with the trip distances for computing the value for cell $\nu_2[\{c_1, c_2\}][\{u_1u_2u_3\}]$ in ν_2 (Table 5(c)).

Table 11: Candidate combined combinations with trip distances for cell $\nu_2[\{c_1, c_2\}][\{u_1u_2u_3\}]$

Combined Combinations	Distances	Total
$\nu_2[\{c_1, c_2\}][\{u_1u_2\}]$+$\nu_0[\emptyset][\{u_3\}]$	144.90 + 68.84	213.74
$\nu_1[\{c_1\}][\{u_1u_2\}]$+$\nu_1[\{c_2\}][\{u_3\}]$	144.90 + 68.84	213.74
$\nu_1[\{c_2\}][\{u_1u_2\}]$+$\nu_1[\{c_1\}][\{u_3\}]$	144.88 + 123.61	268.49
$\nu_0[\emptyset][\{u_1u_2\}]$+$\nu_2[\{c_1, c_2\}][\{u_3\}]$	144.88 + 123.61	268.49

Similarly we compute all combined member columns of ν_0 to ν_4. The rightmost cell of the final table ν_m, which is $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1u_2u_3u_4\}]$ in our example scenario, contains the minimum total trip distance of four trips T_1, T_2, T_3 and T_4, where the trips correspond to users u_1, u_2, u_3 and u_4, respectively. These trips together visit all required POI types $\{c_1, c_2, c_3, c_4\}$ and each POI type is visited by a single user. This is actually the minimum total trip distance of the group for the dynamic scheduling based on the retrieved initial POIs: $p_1^1, p_1^2, p_2^1, p_3^1, p_4^1$. The minimum total trip distance 295.53 is stored in cell $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1u_2u_3u_4\}]$.

Note that the rightmost cell of the final table $\nu_4[\{c_1, c_2, c_3, c_4\}][\{u_1u_2u_3u_4\}]$ contains the minimum total trip distance of the group which is $AggTripDist$ that we have mentioned in Section 5.2. To get the values of T_{min_i} and T_{max_i} for each user u_i, we simply take the minimum and maximum values from Table 5(a) and Table 5(e), respectively. T_{min_i} and T_{max_i} values for users $\{u_1, u_2, u_3, u_4\}$ are

$\{51.55, 93.33, 68.84, 81.78\}$ and $\{51.90, 96.28, 123.69, 90.71\}$, respectively. Using these values we refine the search region based on Theorems 1 and 2. For user u_1, based on Theorem 1, the major axis for the elliptic region E_1 is 51.90. On the other hand, based on Theorem 2, the major axis is $295.53 - (93.33 + 68.84 + 81.78) = 51.58$. We take the best bound among them which is 51.58, the second one.

Each cell of ν_0, ν_1, ν_2, ν_3 and ν_4 also stores the set of POIs for which the minimum trip distance is obtained. For the sake of clarity we do not show them in the tables.

5.5 Algorithms

```
Algorithm 1: $GTS\_Approach(S, D, \mathbb{C})$
input  : $S$, $D$, $\mathbb{C}$
output: A set of trips, $T$
$Initialize()$;
$InitDynTables(|S|, |\mathbb{C}|, \mathcal{V})$;
$ComputeTable(\nu_0)$;
$Enqueue(Q_p, root, MinD(G, root))$;
while $Q_p$ is not empty do
    if $end = 1$ then
        break;
    $\{p, d_{min}(p)\} \leftarrow Dequeue(Q_p)$;
    $r \leftarrow d_{min}(p)$;
    if $p$ is not a POI then
        foreach child node $p_c$ of $p$ do
            $Enqueue(Q_p, p_c, MinD(G, p_c))$;
    else if $\tau(p) \in \mathbb{C}$ and $p \in \bigcup_{i=1}^{n} E_i$ then
        $P \leftarrow InsertPOI(p)$;
        if $init = 0$ and $CheckInclude(P, \mathbb{C})$ then
            $ComputeTrip(S, D, \mathbb{C}, P, \mathcal{V})$;
            $init \leftarrow 1$;
            $isup \leftarrow true$;
        else if $init = 1$ then
            $isup \leftarrow UpdateTrip(\tau(p), S, D, \mathbb{C}, p, \mathcal{V})$;
    if $isup = true$ and $init = 1$ then
        $\{T, Mx, Mi\} \leftarrow UpDynTables(|S|, |\mathbb{C}|, \mathcal{V})$;
        $allipregions \leftarrow UpEllipticRegions(T, Mx, Mi)$;
    if $IsInCircle(G, r, ellipregions)$ then
        $end \leftarrow 1$;
return $T$
```

Algorithm 1 shows the pseudocode of our approach to evaluate GTS queries for both Euclidean space and road networks. It takes the set of source and destination locations, S and D, respectively for a group of n members and the set of required m POI types $\mathbb{C}$ as input. The output is the set of n scheduled trips $T = \{T_1, T_2, \ldots, T_n\}$, where n trips together visit all POI types in $\mathbb{C}$ and no POI type is visited by more than one trip.

As the first step, using function $Initialize()$, Algorithm 1 initializes G to the geometric centroid of source and destination locations, a priority queue Q_p to $\emptyset$, and other variables as follows: $r = 0$, $end = 0$, $isup = false$, and $init = 0$. The variable r represents the radius of current known region. Flags end and $isup$ indicate whether the terminating condition is true and a user's trip has been updated, respectively. Variable $init$ is used to keep track between compute and update trip operations. $Initialize()$ also declares n elliptic regions for n users as $ellipregions = \{E_1, E_2, \ldots, E_n\}$, where the foci of each ellipse E_i is initialized to the source and destination locations of a user and the length of the major axis is set to ∞.

Function $InitDynTables(|S|, |\mathbb{C}|, \mathcal{V})$ initializes the set of dynamic tables $\mathcal{V} = \{\nu_0, \nu_1, \ldots, \nu_m\}$. After that $ComputeTable(\nu_0)$ computes the values for single member columns and combined member columns of the first dynamic table ν_0. The stored trip distances in ν_0 are Euclidean distances if the GTS is query is processed in the Euclidean space, and they are road network distances, otherwise.

The algorithm starts searching from the $root$ of the R^*-tree and inserts the $root$ with $MinD(G, root)$ into a priority queue Q_p. Q_p stores its elements in order of their minimum distances from G, $d_{min}(p)$ that are determined by Function $MinD(G, p)$. For both Euclidean space and road networks, $MinD(G, p)$ returns the minimum Euclidean distance between G and p, where p represents a POI or a minimum bounding rectangle of a R^*-tree node. After that the algorithm removes an element p along with $d_{min}(p)$ from Q_p. At this step, the algorithm updates r, the radius of current known region. If p represents a R^*-tree node, then algorithm retrieves its child nodes and enqueues them into Q_p. On the other hand, if p is a POI then it is added to candidate POI set P, if the POI type is specified in $\mathbb{C}$ and falls inside any user's ellipse E_i. The algorithm uses function $\tau(p)$ to determine the POI type of a POI p.

Function $CheckInclude(P, \mathbb{C})$ checks whether the POI set P contains at least one POI from each POI type in $\mathbb{C}$. When the initial POI set has been found, Function $ComputeTrip(S, D, \mathbb{C}, P, \mathcal{V})$ computes possible trips for all users and populates the single member columns of ν_1 to ν_m using our dynamic programming technique. The algorithm sets $init$ to 1 and $isup$ to $true$. As mentioned before, the stored trip distances in the dynamic tables are Euclidean distances if the GTS is query is processed in the Euclidean space, and they are road network distances, otherwise.

After computing the trips from the initial POI set, if the algorithm retrieves any new POI p, it uses Function $UpdateTrip(\tau(p), S, D, \mathbb{C}, p, \mathcal{V})$ to compute new trips using p and update the single member columns of ν_1 to ν_m, if new trips can improve the stored trip distances in the tables. The function also updates $isup$ accordingly.

If $isup$ is true and the initial set is already found (i.e., $init = 1$), Function $UpDynTables(|S|, |\mathbb{C}|, \mathcal{V})$ updates combined member columns of tables from ν_1 to ν_m based on the logic described in Section 5.4. The function takes n, m and the set of all dynamic tables $\mathcal{V}$ as input, updates the combined member columns of the dynamic tables and returns T, Mx and Mi, where T represents the scheduled trips, Mx and Mi represent the sets $\{T_{max_1}, \ldots, T_{max_n}\}$ and $\{T_{min_1}, \ldots, T_{min_n}\}$, respectively. T_{max_i} and T_{min_i} for $1 \leq i \leq n$ are defined in Section 5.2.

Then using $UpEllipticRegions(T, Mx, Mi)$, the algorithm updates the elliptic bound for all n users, where $ellipregions$ represents the elliptic search regions of the users. The bounds for the elliptic search regions are determined using both Theorem 1 and 2. The algorithm checks the terminating condition of our GTS queries using Function $IsInCircle(G, r, ellipregions)$. This function checks whether all n elliptic search regions is included by the current circular known region or not. If the terminating condition is true, the algorithm updates the terminating flag *end* to 1. At the end of the algorithm, it returns scheduled trips T for n users that provide the minimum total distance.

6 A Straightforward Approach

To the best of our knowledge, we introduce GTS queries in spatial databases and thus, there exists no approach to process GTS queries in the literature. To validate the efficiency of our proposed approach in experiments, we develop a straightforward approach for processing GTS queries, S-GTS, using existing trip planning algorithms.

A straightforward way to process a GTS query would be independently evaluating optimal trips for every group member and for all possible combinations of POI types, and then selecting n trips that together satisfies the conditions of GTS queries and provides the minimum total trip distance for the group. This approach requires multiple independent searches into the database and accesses same POIs multiple times.

```
Algorithm 2: S-GTS_Approach(S, D, ℂ)
input  : S, D, ℂ
output: A set of trips, T
m ← |ℂ|;
n ← |S|;
InitDynTables(|S|, |ℂ|, 𝒱);
ComputeTable(ν_0);
for group member u_i do
    for g ← 1 to m do
        foreach member t_c of ^ℂC_g do
            ν_g[t_c][{u_i}] ← GTP(s_i, d_i, t_c);
{T, Mx, Mi} ← UpDynTables(n, m, 𝒱);
return T
```

Algorithm 2 shows the pseudocode of the S-GTS approach to evaluate GTS queries in the Euclidean and road network spaces. It takes the following parameters as input: the set of source and destination locations, S and D, respectively, for a group of n members and the set of required m POI types $\mathbb{C}$. The output is the set of n scheduled trips $T = \{T_1, T_2, \ldots, T_n\}$, where n trips together visit all POI types in $\mathbb{C}$ and no POI type is visited by more than one trip.

In the first step, Algorithm 2 initializes the dynamic tables ν_0 to ν_m using the function $InitDynTables(|S|, |\mathbb{C}|, \mathcal{V})$, which we mentioned in Section 5.4. After that $ComputeTable(\nu_0)$ computes single member columns and combined member columns of the first dynamic table ν_0. After updating table ν_0, for each member u_i of the group and for each dynamic table ν_g, the algorithm calculates trips for ${}^{m}C_g$ possible sets of POI types using function $GTP(s_i, d_i, t_c)$, and populates the dynamic tables ν_1 to ν_m. The function takes the source and destination locations of u_i, and a set of POI types t_c from $\mathbb{C}$ as input and returns the optimal trip with the trip distance in the Euclidean space or road networks, where the trip starts from s_i, passes through POI types in t_c and ends at d_i. The $GTP(s_i, d_i, t_c)$ function considers all possible orders of POI types in t_c while computing trip distances and returns the minimum one. For the function $GTP(s_i, d_i, t_c)$, any existing trip planning algorithm or group trip planning algorithm (by assuming one group member) can be used. In our experiment, we use the most recent and efficient group trip planning algorithm [7] for this purpose. However, in the S-GTS approach, the function $GTP(s_i, d_i, t_c)$ is called multiple times, and a same POI may be accessed in the database more than once. On the other hand, our GTS approach requires a single traversal on the database and ensures that a single POI is accessed once in the database.

Finally, the algorithm uses the same function $UpDynTables(n, m, \mathcal{V})$ as Algorithm 1 to select the final n scheduled trips for the group. The function updates the combined member columns of the dynamic tables from ν_1 to ν_m, and returns T, and Mx and Mi, where T represents the scheduled trips, Mx and Mi are not used for the S-GTS approach.

Although for the S-GTS approach, we apply the similar dynamic programming that we use for our GTS approach in Section 5, two approaches are different. In the S-GTS approach, we use the dynamic programming technique *once* to find the final scheduled n trips from the already calculated optimal trips of users. On the other hand, the GTS approach incrementally retrieves POIs from the database, calculates the trips of users based on the retrieved POIs, and applies the dynamic programming technique *every time* with the retrieval of a new POI to check whether the new POI can improve the scheduled trips.

7 Experiments

In this section, we evaluate the performance of our approach for processing GTS queries through extensive experiments. Since there is no existing work for GTS queries in the literature, we compare our proposed GTS approach with the straightforward approach (S-GTS) discussed in Section 6 by varying a wide range of parameters.

We evaluate our approach in both Euclidean and road network dataspaces using synthetic and real world datasets. For the real dataset, we used California [2] dataset that contains 87635 POIs of 63 different types. The road network of California has 21048 nodes and 21693 edges. We generated the synthetic datasets of POIs of different types using the uniform random distribution. The whole data space is normalized to 1000x1000 sq. units for both real and synthetic datasets. An R^*-tree is used to store all the POIs of a dataset and a in-memory graph data structure is used to store the road network.

We performed several set of experiments by varying the following parameters: (i) the group size n, (ii) the number of specified POI types m in a GTS query, (iii) the query area A, i.e., the minimum bounding rectangle covering the source and destination locations, and (iv) the dataset size d_s (only in the Euclidean space).

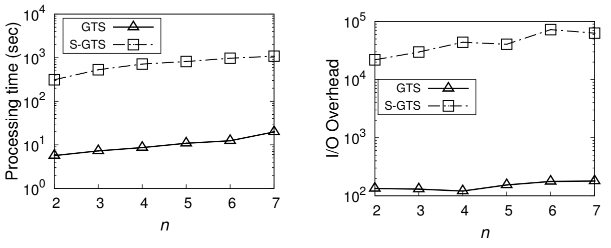
Table 12: Parameter settings

Parameter	Values	Default
Group size(n)	2, 3, 4, 5, 6, 7	3
Number of POI types (m)	2, 3, 4 , 5 , 6	4
Query area(A) (in sq. units)	50x50, 100x100, 150x150, 200x200, 250x250, 300x300	100x100
Dataset size(d_s) (number of POIs in thousands)	5, 10, 20, 40, 80, 160	-
Dataset distribution	Uniform	-

Table 12 shows the range and default values used for each parameter. To observe the effect of a parameter in an experiment, the value of the parameter is varied within its range, and other parameters are set to their default values. We use an Intel Core i5 machine with 2.30 GHz CPU and 4GB RAM to run the experiments. For each set of experiments, we measure two performance metrics: the average processing time and average I/O overhead (I/O access in R^*-tree). The metrics are measured by running 100 independent GTS queries having random source and destination locations, and then taking the average of processing time and I/O access. Since both GTS and S-GTS approaches require the same amount of storage for storing dynamic tables, we do not show them in our experiments.

7.1 Euclidean Space

Effect of group size (n): Figures 9(a) and 9(b) show the processing time and I/O access, respectively, for our GTS and S-GTS approaches. We observe that both processing time and I/O access slightly increase with the increase of the group size. Our GTS approach requires significantly less processing time and I/O access than the S-GTS approach, which is expected. The S-GTS approach computes the optimal trips for each group member and for every possible combination of POI types independently, and thus, accesses the same POIs multiple times in the database. On the other hand, our GTS approach accesses a POI in the database only once and gradually refines the search regions based on the scheduled trips using the dynamic programming technique.

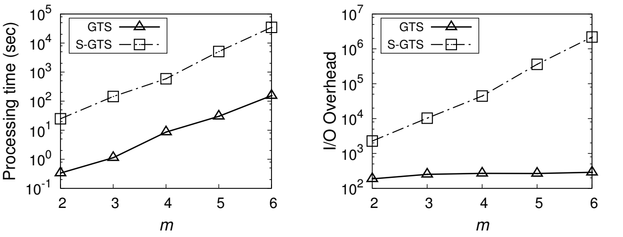


(a) CPU time (b) I/O access

Figure 9: Effect of group size (n) (California dataset)

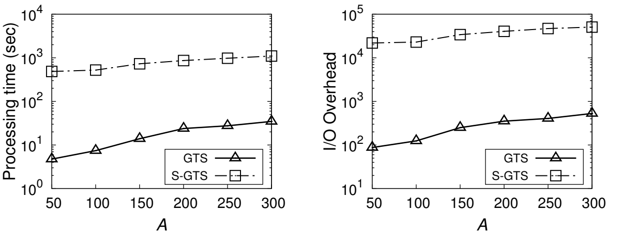
Effect of m***:*** Figures 10(a) and 10(b) show that the processing time and I/O access, respectively, increase with the increase of m. The results show that our GTS approach outperforms the S-GTS approach by a large margin in terms of both I/O access and processing time. Specifically, the improvement for the I/O access is more pronounced for the larger values of m. We observe in Figure 10(b) that the I/Os required by the GTS approach remains almost constant, and the number of I/O access for the S-GTS approach sharply increases with the increase of m. The reason is as follows. For the change of m to $m+1$, the number of independent trip computations in the S-GTS approach for each group member increases by $\sum_{y=0}^{m+1}({}^{m+1}C_y) - \sum_{y=0}^{m}({}^{m}C_y)$, whereas the I/O access of the GTS approach depends on the size of its search region. For an additional POI type, the search region only slightly increases since the $AggTripDist$ and T_{max_i} for any user u_i increase by only a small amount.

Effect of A***:*** Figures 11(a) and 11(b) show experimental results for different values of the query area A. We see that for both approaches, the processing time and I/O access increase with the increase of A. This is because the POI search region becomes large if the source and destination locations are distributed in a large area of the total space. For both metrics, our GTS approach outperforms the S-GTS approach, which is for the similar reasons mentioned for the experiments of varying n.



(a) CPU time (b) I/O access

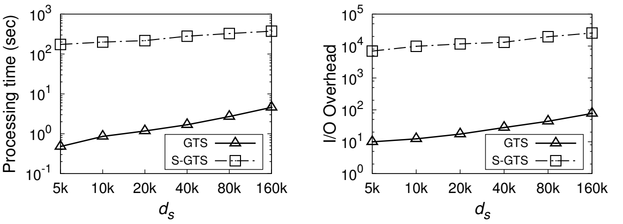
Figure 10: Effect of number of types (m) (California dataset)



(a) CPU time (b) I/O access

Figure 11: Effect of query area (A) (California dataset)

Effect of dataset size (d_s***):*** In this experiment, we examine the performance difference of the two approaches with respect to data set size (d_s). Figures 12(a) and 12(b) show that as the size increases, processing time and I/O access increase for both approaches, which is expected. Like other experiments, the GTS approach takes much less processing time (approx. 192 times) and I/O access (approx. 570 times) than the S-GTS approach for any dataset size.



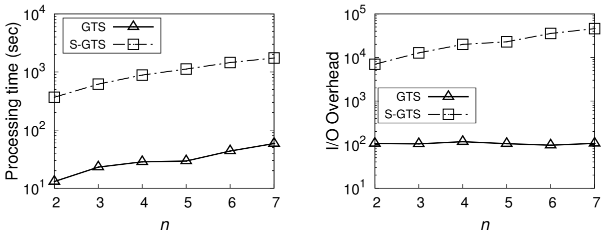
(a) CPU time (b) I/O access

Figure 12: Effect of dataset size (d_s) (Synthetic dataset)

7.2 Road Networks

Experimental results for processing GTS queries in road networks using our proposed approach, GTS, show similar performance and trends like the Euclidean space except that the GTS approach requires on average 6.6 times more query processing time compared to the required processing time in the Euclidean space.

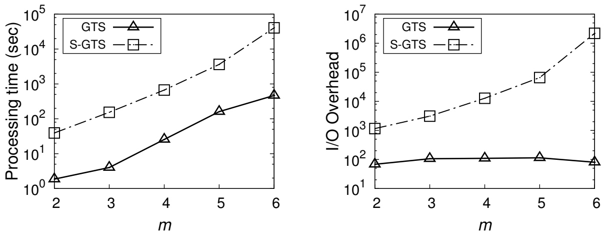
Effect of group size (n): Figures 13(a) and 13(b) show that the query processing time and I/O access increase with the increase of group size n for both approaches, GTS and S-GTS. This is because the number of road network distance computations increase with the increase of n. On the other hand, with the increase of group size n, for our GTS approach, the number of I/O access slightly changes, whereas for the S-GTS approach, the I/O access increases significantly due to the access of same POIs multiple times. For both metrics, the GTS approach outperforms the S-GTS approach.



(a) CPU time (b) I/O access

Figure 13: Effect of group size (n) (California dataset)

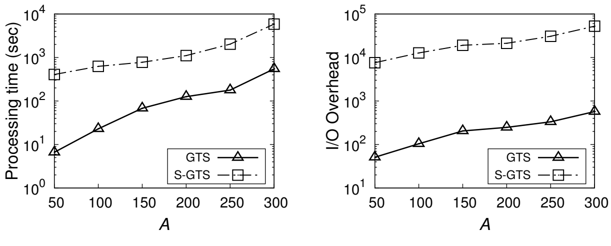
Effect of m***:*** Figures 14(a) and 14(b) show the performance of the GTS approach and the S-GTS approach for varying the total number of POI types m. We observe that the performance trends are similar to those for the Euclidean space. For any number of types, the GTS approach outperforms the S-GTS approach in terms of both I/O access and processing time.



(a) CPU time (b) I/O access

Figure 14: Effect of number of types (m) (California dataset)

Effect of A***:*** Figures 15(a) and 15(b) show that both query processing time and I/O access increase with the increase of A for both approaches, and the GTS approach performs significantly better than the S-GTS approach for both metrics.



(a) CPU time (b) I/O access

Figure 15: Effect of query area (A) (California dataset)

8 Conclusion

In this paper, we have introduced a new type of query, a group trip scheduling (GTS) query in spatial databases that enables a group of users to schedule multiple trips among themselves with the minimum total trip distance of the group members. We propose the first solution to evaluate GTS queries in both Euclidean space and road networks. The refinement technique of the POI search space and the dynamic approach to schedule trips among group members are the key ideas behind the efficiency of our approach. Experiments show that our approach is on average 107 times faster and requires on average 635 times less I/Os than the straightforward approach for the Euclidean space. For road networks, we observed that our approach requires on average 30 times less processing time and 1768 times less I/O access than the straightforward approach. In the future, we aim to protect location privacy [9, 10] of users for GTS queries.

Acknowledgments

This research is supported by Bangladesh University of Engineering and Technology (BUET).

9 References

[1] http://rtreeportal.org/.

[2] California road network data. https://www.cs.utah.edu/~lifeifei/SpatialDataset.htm.

[3] E. Ahmadi and M. A. Nascimento. A mixed breadth-depth first search strategy for sequenced group trip planning queries. In *MDM*, pages 24–33, 2015.

[4] T. Bektas. The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3):209 – 219, 2006.

[5] H. Chen, W. Ku, M. Sun, and R. Zimmermann. The multi-rule partial sequenced route query. In *SIGSPATIAL*, page 10, 2008.

[6] G. Gutin and D. Karapetyan. A memetic algorithm for the generalized traveling salesman problem. *Natural Computing*, 9(1):47–60, 2010.

[7] T. Hashem, S. Barua, M. E. Ali, L. Kulik, and E. Tanin. Efficient computation of trips with friends and families. In *CIKM*, pages 931–940, 2015.

[8] T. Hashem, T. Hashem, M. E. Ali, and L. Kulik. Group trip planning queries in spatial databases. In *SSTD*, pages 259–276, 2013.

[9] T. Hashem and L. Kulik. Safeguarding location privacy in wireless ad-hoc networks. In *Ubicomp*, pages 372–390, 2007.

[10] T. Hashem, L. Kulik, and R. Zhang. Privacy preserving group nearest neighbor queries. In *EDBT*, pages 489–500, 2010.

[11] G. Laporte. A concise guide to the traveling salesman problem. *JORS*, 61(1):35–40, 2010.

[12] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S. Teng. On trip planning queries in spatial databases. In *SSTD*, pages 273–290, 2005.

[13] H. Li, H. Lu, B. Huang, and Z. Huang. Two ellipse-based pruning methods for group nearest neighbor queries. In *GIS*, pages 192–199. ACM, 2005.

[14] H. Li, H. Lu, B. Huang, and Z. Huang. Two ellipse-based pruning methods for group nearest neighbor queries. In *International Workshop on GIS*, pages 192–199, 2005.

[15] J. Li, Q. Sun, M. Zhou, and X. Dai. A new multiple traveling salesman problem and its genetic algorithm-based solution. In *SMC*, pages 627–632, 2013.

[16] Y. Ohsawa, H. Htoo, N. Sonehara, and M. Sakauchi. Sequenced route query in road network distance based on incremental euclidean restriction. In *DEXA*, pages 484–491, 2012.

[17] S. Samrose, T. Hashem, S. Barua, M. E. Ali, M. H. Uddin, and M. I. Mahmud. Efficient computation of group optimal sequenced routes in road networks. In *MDM*, pages 122–127, 2015.

[18] M. Sharifzadeh, M. R. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. *VLDB J.*, 17(4):765–787, 2008.

[19] S. C. Soma, T. Hashem, M. A. Cheema, and S. Samrose. Trip planning queries with location privacy in spatial databases. *World Wide Web*, 2016.

[20] W. Zhou and Y. Li. An improved genetic algorithm for multiple traveling salesman problem. In *Informatics in Control, Automation and Robotics (CAR)*, volume 1, pages 493–495, 2010.

Towards Efficient Maintenance of Continuous MaxRS Query for Trajectories

Muhammed Mas-ud Hussain[1] Goce Trajcevski* [1]
Kazi Ashik Islam[2] Mohammed Eunus Ali[2]

[1]Department of Electrical Engineering and Computer Science
Northwestern University, Evanston, IL, 60208
{mmh683, goce@eecs.northwestern.edu}

[2] Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
{1205007.kai@ugrad.cse.buet.ac.bd, eunus@cse.buet.ac.bd}

ABSTRACT

We address the problem of efficient maintenance of the answer to a new type of query: Continuous Maximizing Range-Sum (Co-MaxRS) for moving objects trajectories. The traditional static/spatial MaxRS problem finds a location for placing the centroid of a given (axes-parallel) rectangle R so that the sum of the weights of the point-objects from a given set O inside the interior of R is maximized. However, moving objects continuously change their locations over time, so the MaxRS solution for a particular time instant need not be a solution at another time instant. In this paper, we devise the conditions under which a particular MaxRS solution may cease to be valid and a new optimal location for the query-rectangle R is needed. More specifically, we solve the problem of maintaining the trajectory of the centroid of R. In addition, we propose efficient pruning strategies (and corresponding data structures) to speed-up the process of maintaining the accuracy of the Co-MaxRS solution. We prove the correctness of our approach and present experimental evaluations over both real and synthetic datasets, demonstrating the benefits of the proposed methods.

1. INTRODUCTION

Recent technological advances in miniaturization of position-aware devices equipped with various sensors, along with the advances in networking and communications, have enabled a generation of large quantities of *(location, time)* data – O(Exabyte) [16]. This, in turn, promoted various geo-social applications where the *(location, time)* information is enriched with (sensed) values from multiple contexts [30, 31]. At the core of many such applications of high societal relevance – e.g., tracking in ecology and environmental monitoring, traffic management, online/targeted marketing, etc. – is the efficient management of mobility data [23].

Researchers in the Spatio-temporal [15] and Moving Objects Databases (MOD) [9] communities have developed a plethora of methods for efficient storage and retrieval of the whereabouts-in-time data, and efficient processing of various queries of interest. Many of those queries – e.g., range, (k) nearest neighbor, reverse nearest-neighbor, skyline, etc. – have had their "predecessors" in traditional relational database settings, as well as in spatial databases [27]. However, due to the motion, their spatio-temporal variants became continuous (i.e., the answer-sets change over time) and even persistent (i.e., answers change over time, but also depend on the history of the motion) [20, 32].

In a similar spirit, this work explores the spatio-temporal extension of a particular type of a spatial query – the, so called, Maximizing Range-Sum query (MaxRS), which can be described as follows:
Q: *"Given a collection of weighted spatial point-objects O and a rectangle R with fixed dimensions, finds the location(s) of R that maximizes the sum of the weights of the objects in R's interior"*.

Various aspects of MaxRS (e.g., scalability, approximate solutions, insertion/removal of points) have been addressed in spatial settings [5, 7, 11, 22, 25, 28] – however, our main

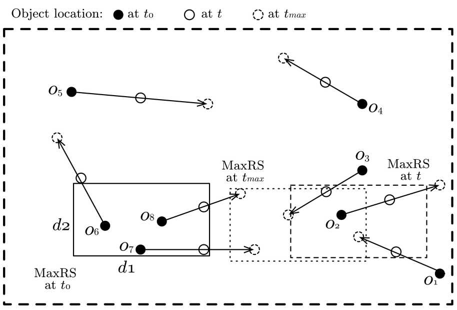


Figure 1: MaxRS vs. Co-MaxRS.

*Research supported by NSF grants III 1213038 and CNS 1646107, and the ONR grant N00014-14-10215.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.36

motivation is based on the observation that there are many application scenarios for which efficient processing of the *continuous* variant of MaxRS is essential. Consider the following query:

Q1: *"What should be the trajectory of a drone which ensures that the number of mobile objects in the Field-of-View of its camera is always maximal?"*.

It is not hard to adapt **Q1** to other application settings: – environmental tracking (e.g., optimizing a range-bounded continuous monitoring of a herd of animals with highest density inside the region); – traffic monitoring (e.g., detecting ranges with densest traffic between noon and 6PM); – video-games (e.g., determining a position of maximal coverage in dynamic scenarios involving change of locations of players/tanks in World of Tanks game). Pretty much any domain involving continuous detection of "most interesting" regions involving mobile entities is likely to benefit from the efficient processing of variants of **Q1** (e.g., mining popular trajectories patterns [33], sports analytics [26], etc.).

Contrary to the traditional *range* query which detects the number of points, or higher dimensionality objects such as (poly)lines and shapes, related to a given *fixed* region, the MaxRS determines the location for placing a given region so that the sum of the weights (i.e., some objective function related to location) is maximized. Originally, the MaxRS problem was tackled by the researchers in computational geometry [11, 22] – however, motivated by its importance in LBS-applications – e.g., best location for a new franchise store with a limited delivery range, most attractive place for a tourist with a restricted reachability bound – recent works have proposed scalable efficient solution for MaxRS in spatial databases [5], including approximate solutions [28] and scenarios where the weights may change and points may be added/deleted [7].

However, the existing solutions to MaxRS queries can only be applied to a specific time instant – whereas **Q1** is a *Continuous MaxRS* (Co-MaxRS) variant. Its weighted-version would correspond to prioritizing certain kinds of mobile objects (e.g., areas with most trucks – by assigning higher weights to trucks) to be tracked by the drone, or certain kinds of tanks in the World of Tanks game. The fundamental difference between MaxRS and Co-MaxRS is illustrated in Figure 1. Assuming that the 8 objects are static at time t_0 and the weights of all the objects are uniform, the placement of the rectangle R indicated in solid line is the solution, i.e., count for optimal R is 3. Other suboptimal placements are possible too at t_0, e.g., covering only o_2 and o_3 with count being 2. However, when objects are mobile, the placement of R at different time instants may need to be changed – as shown in Figure 1 for t_0, t and t_{max}.

A few recent works have tackled the dynamic variants of the MaxRS problem [1, 21]. These works consider objects that may appear or disappear (i.e., insert/delete); however, the locations of the objects do not change over time. To the best of our knowledge, the Co-MaxRS problem has not been addressed in the literature so far and the main contribution of our work can be summarized as follows:

- We formally define the Co-MaxRS problem and identify criteria (i.e., *critical times*) under which a particular MaxRS solution may no longer be valid, or a new MaxRS solution emerges. These, in turn, enable algorithmic solution to Co-MaxRS using procedures executing at discrete time instants.
- Given the worst-case complexity of the problem (consequently, the algorithmic solution), we propose efficient pruning strategies to reduce the cost of recomputing the Co-MaxRS solutions at certain critical times. We present an in-memory data structure and identify properties that enable two such strategies: (1) eliminating the recomputation altogether at corresponding critical time; (2) reducing the number of objects that need to be considered when recomputing the Co-MaxRS solution at given critical times.
- We evaluate our proposed approaches using both real and synthetic datasets, and demonstrate that the pruning strategies yield much better performance than the worst-case theoretical bounds of the Co-MaxRS algorithm – e.g., we can eliminate 80-90% of the critical time events and prune ∼70% objects (on average) when recomputing Co-MaxRS.

In the rest of this paper, Section 2 presents the basic technical background, and Section 3 formalizes the Co-MaxRS problem and describes the basic properties and algorithmic aspects of its solution. Section 4 presents the details of our pruning strategies: properties, data structures and algorithms, and Section 5 presents the quantitative experimental observations illustrating the benefits of the proposed pruning. Section 6 positions the work with respect to the related literature, and Section 7 offers conclusions and directions for future work.

2. PRELIMINARIES

We now review the approaches for solving static MaxRS problem and introduce the concept of kinetic data structures that we subsequently use for solving Co-MaxRS.

2.1 MaxRS for Static Objects

Let $C(p, R)$ denote the region covered by an isothetic rectangle R, placed at a particular point p. Formally:

Definition 1. **(MaxRS)** Given a set O of n spatial points $O = \{o_1, o_2, \dots, o_n\}$, where each o_i associated with[1] a weight w_i , the answer to MaxRS query ($\mathbb{A}_{MaxRS}(O, R)$) retrieves a position p for placing the center of R, such that $\sum_{\{o_i \in (\,O\,\cap\,C(p,R))\}} w_i$ is maximal.

$\sum_{\{o_i \in (\,O\,\cap\,C(p,R))\}} w_i$ is called the *score* of R located at p. If $\forall o_i \in O : w_i = 1$, we have the *count* variant, instances of which at different times are shown in Figure 1. Note that there may be multiple solutions to the MaxRS problem, and in the case of ties – one can be chosen randomly, unless other ranking/preference criteria exist.

Consider the example shown in Figure 2 – the count variant of MaxRS, with a rectangle R of size $d_1 \times d_2$ and five objects (black-filled circles). An in-memory solution to MaxRS (cf. [22]) transforms it into a "dual" *rectangle intersection problem* by replacing each object in $o_i \in O$ by a $d_1 \times d_2$ rectangle r_i, centered at o_i. R covers o_i if and only if its center is placed within r_i. Thus, the rectangle covering the maximum number of objects can be centered anywhere within the area containing a maximal number of intersecting dual rectangles (e.g., $r_3 \cap r_4 \cap r_5$ – gray-filled area in Figure 2).

Using this transformation, an in-memory algorithm to solve the MaxRS problem in $O(n \log n)$ time and $O(n)$ space was devised in [22]. Viewing the top and the bottom edges of each rectangle as horizontal intervals, an *interval tree* – i.e., a binary tree on the intervals – is constructed, and then a

[1] One may also assume that the points in O are bounded within a rectangular area $\mathbb{F}$.

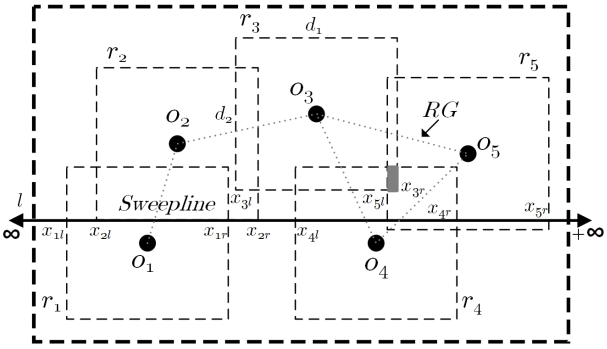


Figure 2: MaxRS → rectangle intersection.

horizontal line is swept vertically, updating the tree at each event. The algorithm maintains the count for each interval currently residing in the tree, where the count of an interval represents the number of overlapping rectangles within that interval. When the sweep-line meets the bottom (top) edge of a rectangle, the corresponding interval is inserted to (deleted from) the interval tree and the count of each interval is updated accordingly. Considering the scenario in Figure 2 and using $[x_{il},\ x_{ir}]$ to denote the left and right boundaries of r_i, when the horizontal sweep-line is at position l, there are 9 intervals: $[-\infty,\ x_{1l}]$, $[x_{1l},\ x_{2l}]$, $[x_{2l},\ x_{1r}]$, $[x_{1r},\ x_{2r}]$, $[x_{2r},\ x_{4l}]$, $[x_{4l},\ x_{5l}]$, $[x_{5l},\ x_{4r}]$, $[x_{4r},\ x_{5r}]$, and $[x_{5r},\ +\infty]$—with counts of 0, 1, 2, 1, 0, 1, 2, 1, and 0 respectively. An interval with the maximum count during the entire sweeping process is returned as the final solution and, since there can be at most $2n$ events (top or bottom horizontal edge of all r_i's) and each event takes $O(\log n)$ processing time, the whole algorithm takes $O(n \log n)$ time to complete.

We note that one may construct a graph RG (*rectangle graph*) where vertices correspond to points/objects in O (i.e., the centers of the dual rectangles) and an edge exists between two vertices o_i and o_j if and only if the corresponding dual rectangles overlap (i.e., $r_i \cap r_j \neq \emptyset$). As illustrated with dotted edges in Figure 2, an area of maximum overlap of dual rectangles corresponds to a maximum clique in RG.

2.2 Kinetic Data Structures

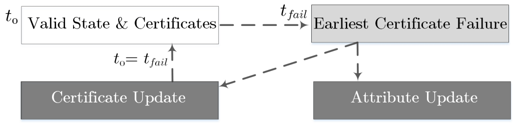


Figure 3: Kinetic Data Structures paradigm.

Kinetic data structures (KDS) [2] are used to track attributes of interest in a geometric system, where there is a set of values (e.g., location – x and y coordinates) that are changing as a function of time in a known manner. To process queries at a (virtual) current time t, an instance of the data structure at initial time t_0 is stored (i.e., values of the attributes of interest), which is augmented with a set of *certificates* proving its correctness at t_0. The next step is to compute the failure times of each certificates – called *events* – indicating that the data structure may no longer be an accurate representation of the state of the system. The events are stored in a priority queue sorted by their failure times. To advance to a time t $(= t_0 + \delta)$, we have to pop all the events having failure times $t_{fail} \le t_0 + \delta$ from the queue in-order, and perform two operations at each event: (1) modify the data structure so that it is accurate at t_{fail} (attribute update), and (2) update the related certificates accordingly (see Figure 3). In this paper, we utilize KDS to maintain the Co-MaxRS answer-set over time and only perform certain tasks at the critical times (events) when a current MaxRS solution may change.

3. BASIC CO-MAXRS

Interval tree was used as in-memory data structure of the planesweep algorithm in both [22] and the subsequent work addressing scalability [5]. However, these techniques cannot be straightforwardly extended to maintain MaxRS solutions continuously – i.e., one cannot expect to have an uncountably-infinite amount of interval trees (at each instant of objects' motion). As it turns out, the answer to Co-MaxRS can change only at discrete time-instants, which we address in the sequel.

Throughout this section, without loss of generality, we assume that each object moves along a single straight line-segment and all the objects start and finish their motion in the same time instant. We will lift this assumption and discuss its impact in Section 4.3.

Continuous MaxRS (Co-MaxRS) is defined as follows:

Definition 2. **(Co-MaxRS)** Given a set O_m of n 2D moving points $O_m = \{o_1,\ o_2,\ \ldots,\ o_n\}$, where each is associated with a trajectory[2] $o_i = [(x_{i1}, y_{i1}, t_{i1}),\ \ldots,\ (x_{i(k+1)}, y_{i(k+1)}, t_{i(k+1)})]$ and a weight w_i; and a time-interval $T = [t_0, t_{max}]$, the answer to Co-MaxRS ($\mathbb{A}_{Co\text{-}MaxRS}(O_m, R, T)$) is a (time-ordered) sequence of pairs $[(l^1_{obj}, [t_0, t_1)), (l^2_{obj}, [t_1, t_2)), \ldots, (l^c_{obj}, [t_{c-1}, t_{max}))]$, where $(l^i_{obj}, [t_{i-1}, t_i))$ denotes the set of objects that determine the possible location(s) for R that is a MaxRS at any time instant $t_j \in [t_{i-1}, t_i) (\subseteq T)$.

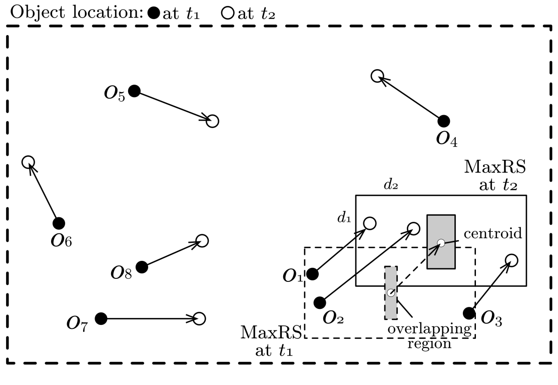
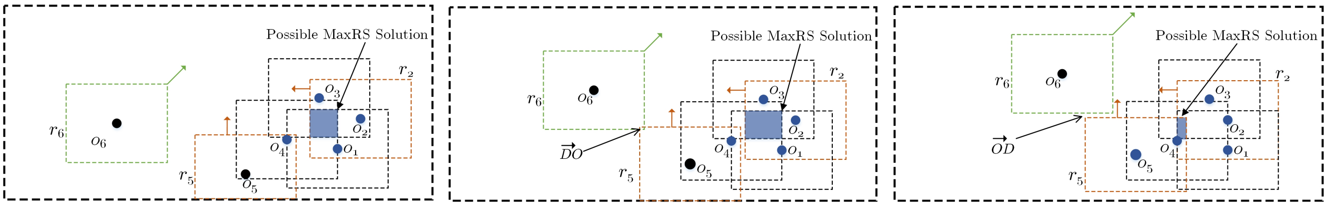


Figure 4: MaxRS location changes from t_1 to t_2, although the objects in the solution are the same.

Note that, instead of maintaining a centroid-location (equivalently, a region) as a Co-MaxRS solution, we maintain a list of objects that are located in the interior of the optimal rectangle placement. The rationale is two-fold: (1)

[2] Again, the trajectories may be bounded within a rectangular area $\mathbb{F}$.



(a) (b) (c)

Figure 5: Co-MaxRS answer can only change when two rectangles' relationship changes from overlap to disjoint (or, vice-versa). Object locations at: (a) t_1 (b) t_2 (c) t_3.

Even for small object movements, the optimal location of the query rectangle can change while objects participating in the MaxRS solution stay the same; and (2) We can easily determine the trajectory (one of the uncountably-many) of the centroid of R throughout the time-interval during which the same set of objects constitutes the solution. An example is shown in Figure 4. At time t_1, objects o_1, o_2, and o_3 fall in the interior of the MaxRS solution. At t_2, although the same objects constitute the MaxRS solution, the optimal location itself has shifted due to the movement of the objects. Suppose there are s objects in the list l^j_{obj} at a particular time instant $t_s \in [t_{j-1}, t_j)$. Given l^j_{obj}, one can find the intersection of the s dual rectangles to retrieve the (boundaries of the possible) location for R at t_s in $O(s)$ time.

We can readily consider an alternative way of representing the Co-MaxRS solution – namely, as a trajectory of the (placement of the) centroid of R. Consider any time interval during which the same set of objects constitutes the solution – e.g., again $(l^j_{obj}, [t_{j-1}, t_j))$. Let $\{o^j_1, \ldots, o^j_s\}$ denote the actual objects from O_m defining l^j_{obj}. Their respective dual rectangles, $\{r^j_1, \ldots, r^j_s\}$ have a common intersecting region at $t = t_{j-1}$ – which, by assumption, is an axes-parallel rectangle. Every point in $\cap^{i=s}_{i=1} r^j_i$ can be a centroid of R covering l^j_{obj} at t_{j-1}. Similarly for $t = t_j$ – once again we have an intersection of the s objects yielding an axes-parallel rectangle, except that both its size and location are changed with respect to the one at $t = t_{j-1}$. The key observations are:
(1) Each r^j_i dual rectangle, when moving along a straight line-segment (to follow the o^j_i) between t_{j-1} and t_j, "swipes" a volume corresponding to a sheared box/parallelopiped.
(2) At each $t \in [t_{j-1}, t_j)$ the intersection of the dual rectangles is non-empty (otherwise, it would contradict the fact that the objects in l^j_{obj} define the solution) and is a rectangle, thereby ensuring that the intersection of the parallelopipeds is continuously non-empty and, again, convex.

Thus, given the $\mathbb{A}_{MaxRS}(O, R)$ at $t = t_{j-1}$ and $t = t_j$, we can simply pick a point in the interior of each of the two (horizontal) rectangles in the $(X, Y, Time)$ space, and the line-segment connecting them is one of the possible trajectories of the centroid of R as the solution/answer-set $\mathbb{A}_{Co\text{-}MaxRS}(O, R, T)$ (of course, for $T = [t_{j-1}, t_j)$).

We now describe how to identify when a recomputation of the MaxRS may (not) be needed due to the possibility of a change in the solution. Consider the example in Figure 5 with 6 objects: $\{o_1, o_2, \ldots, o_6\}$. Let r_i denote the dual rectangle for an object o_i. For simplicity of visualization, assume that only o_2, o_5 and o_6 are moving: o_2 in west, o_5 in north direction (orange rectangles and arrows), and o_6 in the northwest direction. Figure 5a, shows the locations of objects at t_1 and the current MaxRS solution, $l_{obj} = \{o_1, o_2, o_3, o_4\}$ (blue colored objects in Figure 5a). In this setting, r_2 and r_5 do not overlap. Figure 5b shows the objects' locations and their corresponding rectangles at t_2 ($> t_1$). Due to the movement of o_2 and o_5, the maximum overlapped area changed at t_2 (blue-shaded region). But, as r_2 and r_5 still do not overlap, the objects comprising the MaxRS solution are still the same as t_1. Finally, Figure 5c represents the objects' locations at a later time t_3, where r_2 and r_5 are overlapping. This causes a change in the list of objects making up the MaxRS solution, and o_5 is added to the current solution. We note that the solution changed only when two disjoint rectangles began to overlap. If we consider the example in reverse temporal order, i.e., assuming $t_3 < t_2 < t_1$, then the MaxRS solution changed when two overlapping rectangles became disjoint.

Observation: The solution of Co-MaxRS changes only when two rectangles change their topological relationship from *disjoint* to *overlapping* ($\vec{DO}$), or from *overlapping* to *disjoint* ($\vec{OD}$). We consider the objects along the boundary of the query rectangle R as being in its interior, i.e., rectangles having partially overlapping sides and/or overlapping vertices are considered to be *overlapping.* In the rest of the paper, if we need to indicate an occurrence of $\vec{DO}$ or $\vec{OD}$ at a specific time instant t and pertaining to two specific objects o_i and o_j, we will extend the signature of the notation by adding time as a parameter and index the objects in the subscript (e.g., $\vec{DO}_{i,j}(t)$ or $\vec{OD}_{i,j}(t)$).

Thus, as the objects (resp. dual rectangles) move, there are two kinds of changes:
(1) *Continuous Deformation:* As the locations of the rectangles change, the overlapping rectangular regions may change, but the set of objects determining any overlapping rectangular region remains the same.
(2) *Topological Change:* Due to the movement of the rectangles, a $\vec{DO}$ or $\vec{OD}$ transition occurs for a pair of rectangles.

We note that, while the change of the topological relationship is necessary for a change in the answer set in the continuous variant of $\mathbb{A}_{MaxRS}(O_m, R)$ – it need not be sufficient. As shown in Figure 5, the relationship between r_5 and r_6 transitioned from disjoint, to overlap, and to disjoint again. However, none of those changes affected the $\mathbb{A}_{Co\text{-}MaxRS}(O_m, R, T)$ between t_1 and t_3.

In Section 4.3 we will use this observation when investigating the options of pruning certain events corresponding to changes in topological relationships. At the time being, we summarize the steps for a brute-force algorithm for calculating the answer to Co-MaxRS:

Algorithm 1 Basic Co-MaxRS

```
Input: $(O_m, R, T = [t_0, t_{max}])$
Calculate all the time instants for all the pairwise topological changes for the objects in $O_m$
Sort the times of topological changes
For each such time $t_i^{tc}$, execute $\mathbb{A}_{MaxRS}(O, R)$
if Objects defining the answer set are the same then
    Extend the time-interval of the validity of the most recent entry in $\mathbb{A}_{Co\text{-}MaxRS}$ $(O_m, R, T = [t_0, t_{max}])$
else
    Close the time-interval of validity of the prior most-recent entry
    Add a new element into $\mathbb{A}_{Co\text{-}MaxRS}$ $(O_m, R, T = [t_0, t_{max}])$ consisting of the objects defining the $\mathbb{A}_{MaxRS}(O, R)$ at $t_i^{tc}$, with the interval $[t_i^{tc}, t_{i+1}^{tc})$
end if
return $\mathbb{A}_{Co\text{-}MaxRS}(O_m, R, T)$
```

Clearly, the complexity of Algorithm 1 is $O(n^3 \log n)$ – which can be broken into: – $O(n^2)$ for determining the (pairwise) times of topological changes; – $O(n^2 \log n^2)$ for sorting those times; – executing $O(n^2)$ times the instantaneous $\mathbb{A}_{MaxRS}(O, R)$ (at $O(n \log n)$). We note that $O(n^3 \log n)$ is actually a tight worst-case upper-bound, since the solutions in $\mathbb{A}_{MaxRS}(O, R)$ can be "jumping" from one R-region into another that is located elsewhere in the area of interest between any two successive intervals – which are $O(n^2)$.

4. PRUNING IN CO-MAXRS

Given the complexity of the naïve solution – which, again, captures the worst-case possible behavior of moving objects – we now focus on strategies that could reduce certain computational overheads, based on (possible) "localities". We discuss two such strategies aiming to: (1) Reduce the number of recomputations of MaxRS; and (2) Reduce the total number of objects considered when recomputing the MaxRS solution[3], and then present the algorithms that exploit those strategies.

Before proceeding with the details of the pruning, we describe the data structures used.

Figure 6 depicts the data structures used to maintain the Co-MaxRS answer-set based on the KDS framework. Strictly speaking, it consists of:

Object List (OL): A list for storing each object $o_i \in O$, with its current trajectory Tr_{o_i} (i.e., snapshots of location at t_0 and t_{max}), weight w_i, sum of weights of its neighbors in the rectangle graph $WN(o_i)$, and whether or not the object is part of the current MaxRS solution. Note that, o_j is neighbor of o_i if r_i and r_j overlap.

Kinetic Data Structure (KDS): Figure 6 illustrates the underlying KDS (event queue), and its relation with the OL. Each event $E_{i,j}^{t_k}$ is associated with a time t_k, where $t_0 < t_k < t_{max}$. KDS maintains an event queue, where the events are sorted according to the time-value. Each event entry $E_{i,j}^{t_k}$ has pointers to its related objects – two object *identifiers*, and the type of the event – ($\vec{D}O$ or $\vec{O}D$).

Adjacency Matrix (AdjMatrix): Represents the time-dependent rectangle graph RG, with its rows and columns corresponding to the vertices of RG (i.e., the objects from O_m). For each pair of objects o_i and o_j, and a particular (critical) time instant, the *AdjMatrix*$[i][j]$ and *AdjMatrix*$[j][i]$ – set to 1 or 0 – indicate whether the two objects are directly connected with an edge in RG (i.e., their dual rectangles overlap).

4.1 Pruning KDS Events

Recall that the solution to MaxRS problem is equivalent to retrieving the maximum clique in the rectangle graph RG (cf. Section 2). For our first kind of pruning methodology, we leverage on the fact that a KDS event involving two objects o_i and o_j – which can be either $\vec{D}O_{ij}$ or $\vec{O}D_{ij}$ – is equivalent to adding or deleting an edge only between r_i and r_j, and no other objects/rectangles are involved. The properties that allow us to filter out $\vec{D}O$ and/or $\vec{O}D$ types of events without recomputing the MaxRS are discussed next.

$\boldsymbol{\vec{D}O}$: Let $WN(o_i)(t)$ denote the current sum of the weights of the neighbors of an object o_i at time t, and let $score_{max}(t) = score((\mathbb{A}_{MaxRS}(O, R)), t)$ denote the score of the current MaxRS solution at t. During a $\boldsymbol{\vec{D}O}$ event, the lower bound of a MaxRS solution is $score_{max}(t)$, and upper bound of the score (i.e., maximum possible score) of an overlapping region including an object o_i is $(WN(o_i) + w_i)$.

LEMMA 1. *Consider the event $\vec{D}O_{i,j}$ for two objects o_i and o_j, occurring at time $t_{i,j}$. Let $l_{obj}^{(t_{i,j}-\delta)}$ (for some small δ) denote the Co-MaxRS solution just before $t_{i,j}$. After updating $WN(o_i)$ and $WN(o_j)$ at $t_{i,j}$ (i.e., because of $\vec{D}O_{i,j}$), $l_{obj}^{(t_{i,j}-\delta)}$ remains a MaxRS if one of the following two inequalities holds:*

(1) $WN(o_i)(t_{i,j}) + w_i \le score_{max}(t_{i,j} - \delta)$

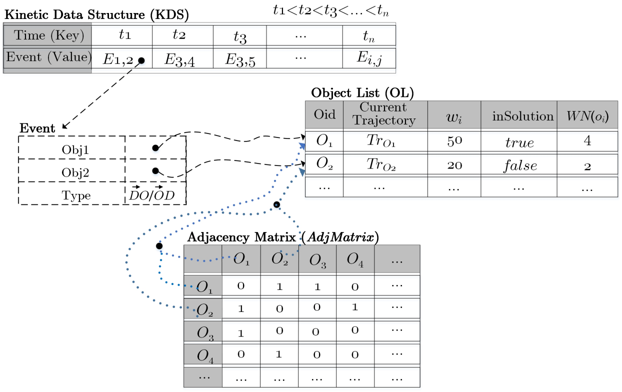
(2) $WN(o_j)(t_{i,j}) + w_j \le score_{max}(t_{i,j} - \delta)$

$\boldsymbol{\vec{O}D}$: In this case the intuition is much simpler – the score/count of an instantaneous MaxRS solution can only decrease (or, remain same) during an $\vec{O}D$ event, and if it decreases (i.e., changes), both of the objects involved in the event must have been in l_{obj}. Thus, we have:

LEMMA 2. *Consider the event $\vec{O}D_{ij}$ for two objects o_i and o_j occurring at time $t_{i,j}$. Let $l_{obj}^{(t_{i,j}-\delta)}$ (for some small δ) be the current MaxRS solution before $t_{i,j}$. If one of the following two conditions holds:*

(1) $o_i \notin l_{obj}^{(t_{i,j}-\delta)}$

(2) $o_j \notin l_{obj}^{(t_{i,j}-\delta)}$


Kinetic Data Structure (KDS)
$t_1 < t_2 < t_3 < \ldots < t_n$
Event
Obj1
Obj2
Type
$\vec{D}O/\vec{O}D$
Object List (OL)
Adjacency Matrix (*AdjMatrix*)

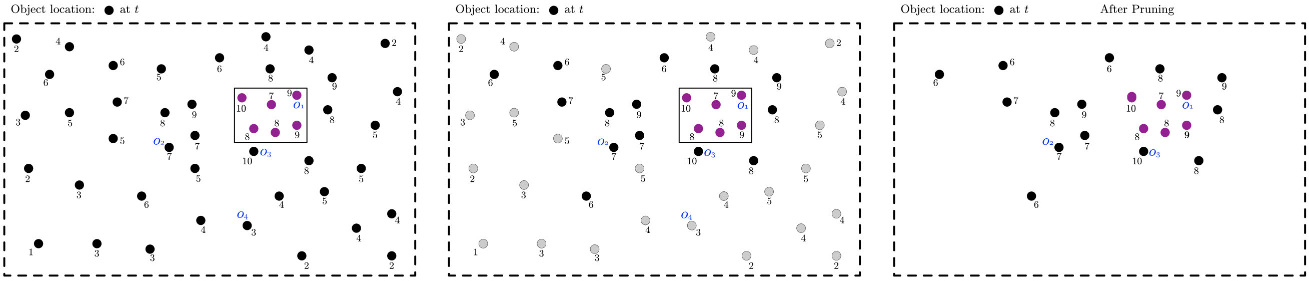
Time (Key)	t_1	t_2	t_3	...	t_n
Event (Value)	$E_{1,2}$	$E_{3,4}$	$E_{3,5}$	...	$E_{i,j}$

Oid	Current Trajectory	w_i	inSolution	$WN(o_i)$
O_1	Tr_{O_1}	50	*true*	4
O_2	Tr_{O_2}	20	*false*	2
...	...	...	...	...

	O_1	O_2	O_3	O_4	...
O_1	0	1	1	0	...
O_2	1	0	0	1	...
O_3	1	0	0	0	...
O_4	0	1	0	0	...
...	...	...	...	...	...

Figure 6: Data structures used.

[3] Due to a lack of space, we do not present the proofs of the Lemmas in this paper, however, they are available at [17].


Object location: ● at t
Object location: ● at t
Object location: ● at t
After Pruning
o_1
o_2
o_3
o_4

(a) (b) (c)

Figure 7: An example showing the objects pruning scheme: (a) Objects' locations and $WN(o_i)$ values at t (b) Grey objects can be pruned using Lemma 3 in a $\vec{DO}$ event (c) Remaining objects after pruning at a $\vec{DO}$ event.

then $l_{obj}^{(t_{i,j}-\delta)}$ remains a MaxRS solution after $\vec{OD}_{ij}$ (i.e, after $t_{i,j}$).

To utilize Lemma 1 and 2, we maintain for each $o_i \in O_m$ the value of $WN(o_i)$, and whether or not the object is part of the current MaxRS solution. In Figure 6, two variables *inSolution* and $WN(o_i)$ are used for this purpose, updated accordingly during the processing of $\vec{DO}$ and $\vec{OD}$ events.

4.2 Objects Pruning

After filtering out many of the recomputations (Lemma 1 and Lemma 2), it is desirable to reduce the number of objects required in the recomputation. Towards that, we the following observations: (1) $WN(o_i) + w_i$ is an upper bound on possible MaxRS scores containing an object o_i; (2) $score_{max}$, the current MaxRS score, is a lower bound on possible MaxRS scores after a $\vec{DO}$ event; and (3) $score_{max} - min\{w_i, w_j\}$ is a lower bound on possible MaxRS scores after a qualifying $\vec{OD}_{ij}$ event. Let $E_{i,j}$ denote any event involving two objects o_i and o_j (be it $\vec{DO}_{ij}$ or $\vec{OD}_{ij}$). We have:

Lemma 3. *After updating $WN(o_i)$ and $WN(o_j)$ at E_{ij}, an object o_k can be pruned before recomputing MaxRS if one of the following two conditions holds:*

(1) $E_{i,j}$ is a $\vec{DO}$ event and $WN(o_k) + w_k \leq score_{max}$

(2) $E_{i,j}$ is an $\vec{OD}$ event and $WN(o_k) + w_k \leq score_{max} - min\{w_i, w_j\}$

Example 1. Figure 7a demonstrates an example scenario with 46 objects. For the sake of simplicity, we only consider the counting variant (i.e., $\forall o_i \in O : w_i = 1$) in this example. The count of neighbors (i.e., $WN(o_i)$) for each object is shown as a label, and the current MaxRS solution is illustrated by a solid rectangle where $score_{max}$ (or, $count_{max}$) = 6. Members of l_{obj} are colored purple in Figure 7. Some of the objects are marked with an id (e.g., o_1, o_2, o_3, and o_4), so that they can be identified clearly in the text. In this scenario, to process any event, we will first update the appropriate $WN(o_i)$ and *inSolution* values. Then, suppose a new $\vec{DO}$ event is processed for one of the objects for which $WN(o_i) \leq 5$, e.g., between o_3 and o_4. Then that event will be pruned and MaxRS answer-set will remain the same as the maximum possible count of a MaxRS including that object will be $(5+1)$=6. Similarly, any $\vec{OD}$ event involving an object other than the purple ones would be filtered out. Figure 7b illustrates the application of Lemma 3, based on which all the objects in grey can be pruned during a $\vec{DO}$ event before recomputing MaxRS. Thus, after applying Lemma 3, we can prune 26 objects in linear time, i.e., going through the set of objects once and verifying the respective conditions. After pruning, 20 objects will remain (cf. Figure 7b) – only 43% of the total objects.

According to Lemma 1, a $\vec{DO}_{ij}$ event is not pruned when both $WN(o_i) + w_i > score_{max}$ and $WN(o_j) + w_j > score_{max}$ hold. Let us use $N(o_i)$ to denote the list of neighbors of any object o_i. Additionally, we employ $CN(o_i, o_j)$ to represent common neighbors of two objects o_i and o_j, i.e., $N(o_i) \cap N(o_j)$. In this setting, there are two possible cases:

Case 1: *Both $o_i, o_j \notin l_{obj}$.* The observation here is that if there exists a new MaxRS solution at a $\vec{DO}_{ij}$ event, then both o_i and o_j must be present in the new solution as only they are affected by the new $\vec{DO}$ event – all other objects (and their related attributes) remain the same. Additionally, for any MaxRS solution including both o_i and o_j, only the members of $CN(o_i, o_j)$ can be in l_{obj}.

Case 2: *Either $o_i \in l_{obj}$ or $o_j \in l_{obj}$.* Let us assume $o_i \in l_{obj}$. Then, if o_j overlaps with all objects $o_k \in l_{obj}$ (an $O(|l_{obj}|)$ check), then we can directly have a new MaxRS solution including o_j, i.e., $l_{obj} = l_{obj} \cup o_j$. If this check fails, we can follow the similar procedure as case 1. Note that, the case of both $o_i, o_j \in l_{obj}$ is not possible as it contradicts the concept of $\vec{DO}_{ij}$ event, i.e., o_i and o_j are mutually disjoint before $\vec{DO}_{ij}$. Based on the above observations, we have the following two lemmas:

Lemma 4. *For an event $\vec{DO}_{ij}$ involving two objects o_i and o_j, we can prune all the objects except o_i, o_j, and $CN(o_i, o_j)$ before recomputing MaxRS.*

Lemma 5. *For an event $\vec{DO}_{ij}$ involving two objects o_i and o_j where $o_i \in l_{obj}$, we can set $l_{obj} \cup o_j$ as the new MaxRS solution if o_j overlaps with all objects $o_k \in l_{obj}$.*

To take advantage of Lemma 4, we need to keep track of neighbors of all the objects in addition to $WN(o_i)$, which is the purpose of the adjacency matrix (*AdjMatrix* in Figure 6). We note that one could also maintain a list $N(o_i)$ for each object – however, although each approach would incur $O(n^2)$

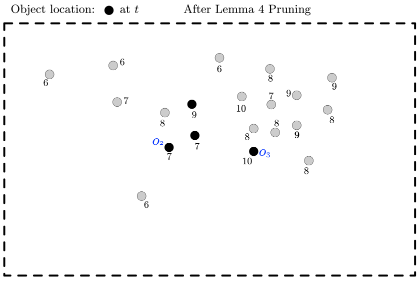


Figure 8: Application of Lemma 4 in $\vec{D}O$ events.

space overhead in the worst case, the adjacency matrix has certain advantages:

- Updating of the matrix information can be done in $O(1)$ time. For example, at a $\vec{D}O_{i,j}$ event we can directly set *AdjMatrix*[i][j]=1 and *AdjMatrix*[j][i]=1. Similarly, *AdjMatrix*[i][j] and *AdjMatrix*[j][i] can be set to 0 at an $\vec{O}D_{i,j}$ event.
- We can compute $CN(o_i, o_j)$ for two objects o_i and o_j efficiently by doing a *bit-wise AND* operation over *AdjMatrix*[i] and *AdjMatrix*[j].

Example 2. Suppose there is a new $\vec{D}O$ event between objects o_1 and o_3 in the example in Figure 7. The event will not be pruned because both $WN(o_1)$ and $WN(o_3) > 5$. As $o_1 \in l_{obj}$, we will first check if o_3 overlaps with all other members of l_{obj} (purple colored objects). As it does overlap with all the members of l_{obj}, we can directly output $l_{obj} \cup o_3$ as the new solution using Lemma 5. On the other hand, suppose the new $\vec{D}O$ occurs between o_2 and o_3. Using Lemma 4, we can prune all the objects except o_2, o_3, and $N(o_2) \cap N(o_3)$. This leaves us with only 4 remaining objects (cf. Figure 8) – 91.3% objects are pruned from the calculation. Obviously, score of the recomputed MaxRS will be less than the $score_{max}$ we already have (i.e., 6), and thus no change to the solution of Co-MaxRS will be made. We can see, Lemma 4 and Lemma 5 greatly optimizes processing of $\vec{D}O$ events.

4.3 KDS Properties and Algorithmic Details

Instead of a single line-segment, moving objects trajectories in practice are often polylines with vertices corresponding to actual location-samples. To cater to this, we introduce another kind of event, pertaining to an individual object – *line-change* event at a given time instant, denoted as $E_{lc}(o_i, t_{l_i})$. Suppose, for a given object o_i, we have $k + 1$ time-samples during the period T as $t_{i1}, t_{i2}, \ldots, t_{i(k+1)}$, forming k line-segments. Note that the frequency of location updates may vary for different objects; even for a single object, the consecutive time-samples may have different time-gap. Initially, we insert the second time-samples for all the objects into the KDS as line-change events (cf. Figure 6). When processing $E_{lc}(o_i, t_{l_i})$ we need to compute: (a) Next $\vec{O}D$ events with the neighbors; and (b) Next $\vec{D}O$ events with other non-neighboring objects. We also need to insert a new line-change event at $t_{l_{(i+1)}}$ for o_i into the KDS. Thus, processing a line-change event takes $O(n)$ time. Note that a particular trajectory may start (appear) and/or finish its trip (disappear) at any time t, where $t_0 < t < t_{max}$ and we can use similar ideas to handle these special cases in $O(n)$ time.

KDS Properties: We proceed with briefly analyzing the properties of our proposed KDS-based structure (in the spirit of [2]), which shows that our adaption of KDS is responsive, efficient, local, and practically responsive.

(1) Number of certificates altered during an event (Responsiveness): Recall that we have two kinds of core events:

$\vec{D}O$ Event: At such an event we need to compute the time of the next $\vec{O}D$ event between the two objects and insert that to KDS if it falls within the given time-period T. Thus, only one new event (certificate) is added.

$\vec{O}D$ Event: For these events, we just need to process them, and no new event is inserted into KDS.

In both cases, the number is a small constant – conforming with the desideratum.

(2) The size of KDS (Compactness): In case of our adaptation of the KDS, we can have at most $O(n^2)$ $\vec{D}O$ and $\vec{O}D$ events at once. If we consider the additional line-change events for the polyline moving objects trajectories, there can be one such event for each object at any particular time, i.e., $O(n)$ such events. Thus, the size of KDS at a particular time is at most $O(n^2)$. However, as we will see in Section 5, in practice the size (total events) can be significantly smaller than this upper-bound – meeting the desideratum, i.e., $O(n^\epsilon)$ for some arbitrarily small $\epsilon > 0$.

(3) The ratio of internal and external events (Efficiency): In our KDS, the $\vec{D}O$ and $\vec{O}D$ events are external events (i.e., possibly causing changes to the Co-MaxRS answer-set), and the line-change events are internal. Thus, the ratio between total number of events and external events is $\frac{O(n^2)+O(n)}{O(n^2)}$, which is relatively small. This is a desired property of an *efficient* KDS [2].

(4) Number of certificates associated with an object (Locality): An object can have $n - 1$ $\vec{D}O$ and $\vec{O}D$ events with the other objects, and 1 line-change event at a particular time instant, i.e., the number of events associated with an object is $O(n)$, which is an acceptable bound.

Subsequently, our adaption of KDS is responsive, efficient, local and, in practice, compact too.

Algorithmic Details: In Algorithm 2, we present the detailed method for maintaining Co-MaxRS for a given time period $[t_0, t_{max}]$. As mentioned, for each object, in addition to *WN* and *inSolution* variables, we also keep track of the active neighbors in RG via *AdjMatrix*. After initialization (line 1 and 2), the KDS is populated with all the initial events that fall within the given time-period (line 3) – a step taking $O(n^2)$ time. Then, we retrieve the current solution, i.e., the list of objects, and create a new time-interval of its validity, starting at t_{start}^{new} in lines 4-6. We update the *inSolution* values of related objects whenever we compute a new MaxRS solution, and discard an old one (lines 7, 15, and 16). Lines 8–19 process all the events in the KDS in order of their time-value, and maintain the Co-MaxRS answer-set throughout. The top event from the KDS is selected and processed using the function *EventProcess* (elaborated in Algorithm 3). After checking whether a new solution has been returned from *EventProcess*, the answer-set is adjusted in the sense of closing its interval of validity (t_{end}^{new})

Algorithm 2 Co-MaxRS (OL, R, t_0, t_{max})

```
 1: $KDS \leftarrow$ An empty priority queue of events
 2: $\mathbb{A}_{Co\text{-}MaxRS} \leftarrow$ An empty list of answers
 3: Compute next event $E_{next}, \forall o_i \in OL$ and push to $KDS$
 4: current $\leftarrow$ Snapshot of object locations at $t_0$
 5: $(loc_{opt}, score_{max}, l_{obj}) \leftarrow$R_Location_MaxRS(current)
 6: $t^{new}_{start} \leftarrow t_0$
 7: Update $inSolution$ variable for each $o_i$ in $l_{obj}$
 8: while $KDS$ not EMPTY do
 9:    $E_{i,j} \leftarrow KDS.Pop()$
10:    $(l'_{obj}, score_{max}) \leftarrow$ EventProcess$(E_{i,j}, KDS, l_{obj},$
       $score_{max})$
11:    if $l_{obj} \neq l'_{obj}$ then
12:       $t^{new}_{end} \leftarrow t_i$
13:       $\mathbb{A}_{Co\text{-}MaxRS}.Add(l_{obj}, [t^{new}_{start}, t^{new}_{end}])$
14:       $t^{new}_{start} \leftarrow t_i$
15:       Update $inSolution$ variable for each $o_i$ in $l_{obj}$
16:       Update $inSolution$ variable for each $o_i$ in $l'_{obj}$
17:       $l_{obj} \leftarrow l'_{obj}$
18:    end if
19: end while
20: $t^{new}_{end} \leftarrow t_{max}$
21: $\mathbb{A}_{Co\text{-}MaxRS}.Add(l_{obj}, [t^{new}_{start}, t^{new}_{end}])$
22: return $\mathbb{A}_{Co\text{-}MaxRS}$
```

Algorithm 3 EventProcess ($E_{i,j}$, KDS, l_{obj}, $score_{max}$)

```
 1: Update $WN(o_i)$, $WN(o_j)$, and $AdjMatrix$ accordingly
 2: if $E_{i,j}.Type = \vec{DO}$ then
 3:    Compute $E_{next}$ for objects $o_i$ and $o_j$
 4:    if $E_{next} \neq$ NULL and $E_{next}.t \in [t_0, t_{max}]$ then
 5:       $KDS.Push(E_{next})$
 6:    end if
 7: end if
 8: if $E_{i,j}.Type = \vec{DO}$ and $(WN(o_i) + w_i \leq score_{max}$ or
    $WN(o_j) + w_j \leq score_{max})$ then
 9:    return $(l_{obj}, score_{max})$
10: end if
11: if $E_{i,j}.Type = \vec{OD}$ and $(o_i.inSolution = false$ or
    $o_j.inSolution = false)$ then
12:    return $(l_{obj}, score_{max})$
13: end if
14: if $E_{i,j}.Type = \vec{DO}$ and Either $o_i/o_j \in l_{obj}$ then
15:    $o_k \leftarrow o_j/o_i$
16:    if $o_k$ and $l_{obj}$ are mutually overlapping then
17:       return $(l_{obj} \cup o_k, score_{max} + w_k)$
18:    end if
19: end if
20: $OL' \leftarrow OL$
21: if $E_{i,j}.Type = \vec{DO}$ then
22:    $CN(o_i, o_j) \leftarrow$ Compute-CN $(AdjMatrix, o_i, o_j)$
23:    $OL' \leftarrow CN(o_i, o_j) \cup \{o_i, o_j\}$
24: end if
25: for all $o_k$ in $OL'$ do
26:    if $(E_{i,j}.Type = \vec{DO}$ and $WN(o_k) + w_k \leq score_{max})$
       or $(E_{i,j}.Type = \vec{OD}$ and $WN(o_k) + w_k \leq score_{max} -$
       $min(w_i, w_j))$ then
27:       Prune $o_k$
28:    end if
29: end for
30: current $\leftarrow$ Snapshot of objects in $OL'$ at $t_i$
31: $(loc'_{opt}, score'_{max}, l'_{obj}) \leftarrow$R_Location_MaxRS(current)
32: if $(E_{i,j}.Type = \vec{OD})$ or $(E_{i,j}.Type = \vec{DO}$ and
    $score'_{max} > score_{max})$ then
33:    return $(l'_{obj}, score'_{max})$
34: end if
35: return $(l_{obj}, score_{max})$
```

which, along with the corresponding l_{obj} are appended to $\mathbb{A}_{Co\text{-}MaxRS}(O_m, R, T)$ (for brevity, the ".Add()" notation is used). A modified version of the MaxRS algorithm from [22] is used where, in addition to the score, the list l_{obj} is also returned – cf. *R_Location_MaxRS* in line 5. Note that, the condition check at line 11 in implementation actually takes constant time, which we detect via setting a boolean variable during MaxRS computation.

The processing of a given KDS event $E_{i,j}$ is shown in Algorithm 3. In line 1, the *WN* of the relevant objects and *AdjMatrix* are updated. In lines 2–7, we compute new $\vec{OD}$ events and update the KDS. Lines 8–13 implement the ideas of Lemma 1 and Lemma 2, which takes $O(1)$ time. Lines 14–19 implement the idea of Lemma 5 to process a special kind of $\vec{DO}$ events. Line 20 introduces a new list OL', which will eventually retain only the unpruned objects. Lines 21-24 employ the idea of Lemma 4 for $\vec{DO}$ events. Lines 25–29 implement the ideas of objects pruning (Lemma 3), which takes $O(n)$ time. Finally, MaxRS is recomputed in lines 30–31 based on the current snapshot of the remaining moving objects in $O(n \log n)$ time (for brevity, we omitted handling line-change events in Algorithm 3). Lines 32–34 ensure that only valid computed values are returned, i.e., when $score'_{max} > score_{max}$ for $\vec{DO}$ events.

Discussion: In the worst-case, Co-MaxRS for n trajectories with k segments throughout the query time-interval, has $O(kn^2)$ events. In KDS, $O(n^2)$ events are added at the beginning, then at each of the $O(kn)$ line change events, $O(n)$ new events may be created, resulting in $O(kn^2)$ events in total. Observe that between two consecutive event-times t_{s-1} and t_s, there is a Co-MaxRS path of constant complexity (i.e., the centroid of R moves along a straight line-segment). As mentioned in Section 3, this follows from the fact that the Co-MaxRS solution covering a particular list l^s_{obj} in the sequence $(\mathbb{A}_{Co\text{-}MaxRS}(O_m, R, T))$ for the interval $[t_{s-1}, t_s]$, is the (maximum) intersection of sheared-boxes generated by the motion of the dual rectangles of the objects in l^s_{obj}. Thus, the worst-case combinatorial complexity of the path of the centroid of the Co-MaxRS solutions is $O(kn^2)$ – with a note that there may be discontinuities between consecutive locations of the centroids (i.e., the solution "jumps" from one location to another). The overall worst-case complexity when considering trajectories with multiple segments (i.e., polyline routes) is $O(kn^3 \log n)$.

We close this section with two notes:

(1) While the worst-case complexity of processing Co-MaxRS is high, such orders of magnitude are not uncommon for similar types of problems – i.e., detecting and maintaining flocks of trajectories [8]. However, as our experiments will demonstrate, the pruning strategies that we proposed can significantly reduce the running time.

(2) A typical query processing approach would involve *filtering* prior to applying pruning – for which an appropriate index is needed, especially when data resides on a secondary storage. Spatio-temporal indexing techniques abound since the late 1990s (extensions of R-tree or Quadtree variants,

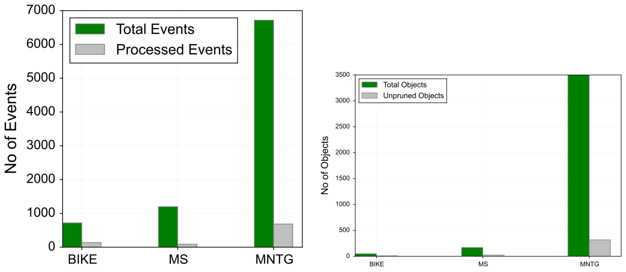
combined subdivisions in spatial and temporal domains, etc. [14, 19]). Throughout this work we focused on efficient in-memory pruning strategies, however, in Section 5 as part of our experimental observations, we provide a brief illustration about the benefits of using an existing index (TPR* tree [29]) for further improving the effects of the pruning. This, admittedly, is not a novel research or a contribution of this work, but it serves a two-fold purpose: (a) to demonstrate that our proposed approaches could further benefit by employing indexing; (b) to motivate further research for appropriate index structure.

5. EXPERIMENTAL OBSERVATIONS

Datasets: We used two real-world datasets and another synthetic one during our experiments. The first real-world dataset we used is the bicycle GPS (BIKE-dataset) collected by the researchers from University of Minnesota [10], containing 819 trajectories from 49 different participant bikers, and 128,083 GPS points. The second one is obtained from [34] (MS-dataset), which contains GPS-tracks from 182 users in a period of over five years collected by researchers at Microsoft with 17,621 trajectories in total, covering 1,292,951 km and over 50,176 hours (with GPS samples every 1-5 seconds). To demonstrate the scalability of our approach, we also used a large synthetic dataset (MNTG-dataset) generated using Minnesota Web-based Traffic Generator [18]. The generated MNTG-dataset consists of 5000 objects, and 50000 trajectories with 400 points each, where we set the option that objects are not constrained by the underlying network. For every object in the synthetic dataset, we generated its weight uniformly in the range from 1 to 50, while weights in Bike-dataset and MS-dataset (real-world datasets) were set to 1.

For each of the dataset used in the experiments, we considered one trajectory per object during a run and we averaged over all the runs to get representative-observations. The default values of the number of objects for BIKE, MS, and MNTG dataset are 49, 169, and 5000 respectively. The query time is set to the whole time-period (lifetime of trajectories) during a particular run for each respective dataset, and the base value of range area (R) for each of the BIKE, MS, and MNTG dataset is 500000, 100000, and 400000 m^2 respectively.

Implementations: We implemented all the algorithms in Python 2.7, aided by powerful libraries, e.g., Scipy, Matplotlib, Numpy, etc. We conducted all the experiments on a machine running OS X El Capitan, and equipped with Intel Core i7 Quad-Core 3.7 GHz CPU and 16GB memory. As no prior works exist that directly deal with the Co-MaxRS problem, we use Algorithm 1 as our baseline for comparison. In addition to the Algorithms 1, 2 and 3, we have two additional implementations[4]: (1) As mentioned at the end of Section 4.3, we added TPR*-tree index, to investigate the further benefits in terms of pruning with KDS; and (2) To demonstrate the benefits of our pruning schemes, we tested them against a trivial approximate-solution to Co-MaxRS: one that would periodically re-evaluate the query throughout its time-interval of interest. In other words, MaxRS is re-computed at each $t + \delta$, i.e., δ is a fixed time-period (default δ=5s).



(a) (b)

Figure 9: (a) Events Pruning (b) Objects Pruning.

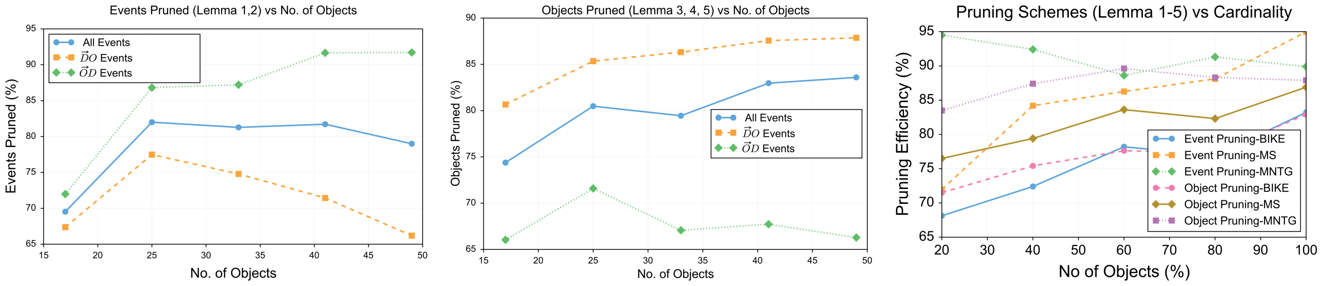
Performance of Pruning Strategies: Our first observations are shown in Figure 9a and they demonstrate the effectiveness of our events pruning strategy over both the real and synthetic datasets. The most amount of pruning is obtained in MS-dataset, while the other two datasets also show more than 80% pruning. Note that, the number of actual recomputation-events are well below the worst-case theoretical upper-bound, e.g., only 103 events are processed for 49 objects (trajectories) running for an hour in Bike-dataset. Similar results are obtained for the objects pruning scheme, as demonstrated in Figure 9b – indicating that the pruning schemes perform nearly equally well in all three datasets.

Impact of Cardinality: Figure 10 illustrates the impact of the cardinality on the effectiveness of our pruning methods. In Figure 10a, from the experiment done on the BIKE-dataset, we can deduce an interesting relation: as the datasize increases, more $\vec{OD}$ kind of events are pruned, whereas (cf. Figure 10b), objects pruning slightly decreases for $\vec{OD}$ as the datasize increases. On the other hand, $\vec{DO}$ events exhibit completely opposite behavior. This, in a sense, neutralizes the overall impact of the increase in cardinality for our pruning scheme. Figure 10c demonstrates the effect of increasing the cardinality of objects on the pruning schemes for all the dataset – hence, the label on the X-axis indicates the percentage of all the objects for the respective datasets.

Influence of Range Size: This experiment was designed to observe the effect of different range sizes, i.e., the area of R – $d_1 \times d_2$ over the pruning strategies. As shown in Figure 11a, increasing range area (the values on X-axis indicate multiples of the base-size for each dataset) results in fewer portion of events pruned. This occurs because as the area of R grows, there are more overlapping dual rectangles among the moving objects. Similarly, the growing rectangle size had adverse effects on the objects pruning scheme as well (cf. Figure 11b). We note, though, that even with quite large values of R (e.g., 50000 m^2) we have more than 60% of pruning through our proposed methods.

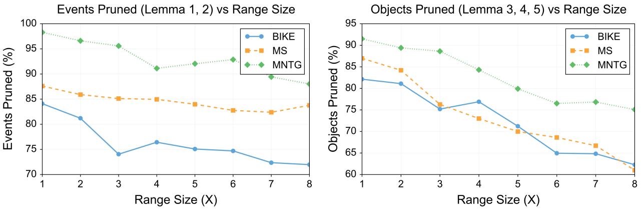
Benefits of indexing: Indexing the trajectories provides a filtering power which can be used as an additional pruning benefits (with respect to the Lemmas in Section 4) in terms of retrieving overlapping neighbors for any object. Figure 12 demonstrates benefits of indexing over varying cardinality (experiment done on MNTG Dataset). The running time using index is almost 100% times faster (half) for 5000 objects. We re-iterate that, as mentioned in Section 4.3, this is not a research contribution of the paper but only serves the purpose to demonstrate that an index is likely to yield

[4] We note that all the datasets and the source code of the implementation are publicly available at http://www.eecs.northwestern.edu/∼mmh683.



(a) (b) (c)

Figure 10: Impact of cardinality on the pruning schemes: (a) Different events pruning (BIKE-dataset) (b) Objects pruning (BIKE-dataset) (c) Overall objects and events pruning (all datasets).



(a) (b)

Figure 11: (a) Events pruning strategy; (b) Objects pruning strategy against varying range sizes.

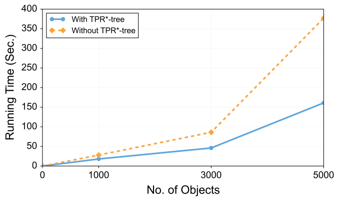


Figure 12: Potential impact of index.

further benefits for our proposed approaches.

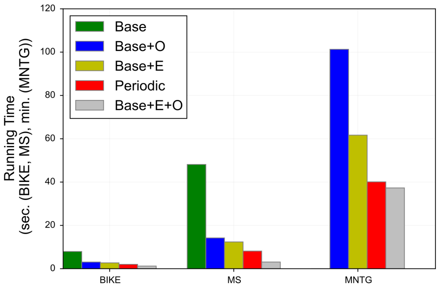
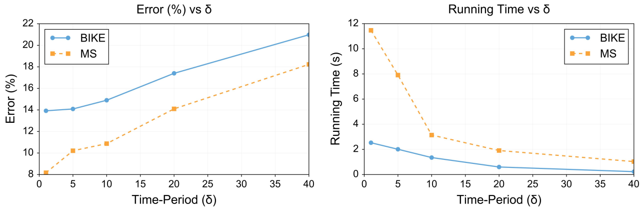


Figure 13: Running-time in different datasets.

Running Time Comparison: We ran the algorithms over the three datasets and the result is shown in Figure 13. This is the first experiment in which we also report observations regarding the periodical processing of the MaxRS – and it serves the purpose to provide a complementary illustration of the benefits of our methodologies. Namely, even if one is willing to accept an error in the result and perform only periodic snapshot MaxRS, our pruning techniques are still more efficient, while ensuring correct/complete answer set. The *Base*, *(Base+O)*, *(Base+E)*, *(Base+E+O)*, and *Periodic* in Figure 13 denote the base Co-MaxRS, base + objects pruning, base + events pruning, base + both events and objects pruning, and periodical processing of MaxRS (δ=5s), respectively. In case of MNTG-dataset, the average running time (for a set of trajectories) is shown in minutes, while for the other two datasets the unit it is shown in seconds. We omitted the average running time for the base algorithm over MNTG-dataset in Figure 13 which is more than 10 hours (to avoid skewing the graph). The base Co-MaxRS is the slowest among these algorithms, as it recomputes MaxRS at each event. The effect of both events and objects pruning schemes on running time is prominent, although events pruning exhibits a bigger impact individually (preventing unnecessary recomputations). When both pruning strategies are applied together, the algorithm speeds-up significantly – almost 6-15 times faster than the base algorithm over all the datasets – making it the fastest among all the evaluated algorithms.



(a) (b)

Figure 14: Impact of δ on (a) Error (b) Running Time of periodic-MaxRS.

Periodical Processing: The last observations illustrate the errors induced by periodical processing of MaxRS (periodic-MaxRS) to approximate Co-MaxRS. Note that we exclude performing periodic-MaxRS related experiments on the large synthetic dataset (MNTG-dataset) as the correctness, rather than scalability, is a concern. In Figure 14, the impact of (δ) is illustrated both on running time and correctness. As δ increases the error in the approximation increases as well. Even for a small δ (e.g., $1s$), the respective error is still around 8-14% (cf. Figure 14a). Complementary to this, in Figure 14b, we see that as δ decreases, the running time increases too. For both Bike-dataset and MS-dataset, for small δ values ($\leq 5s$), average processing time is much longer than our proposed algorithm (*Base+E+O*) and yet it contains errors.

6. RELATED WORKS

The problem of MaxRS was first studied in the Computational Geometry community, with [11] proposing an in-memory algorithm to find a maximum clique of intersection graphs of rectangles in the plane. Subsequently, [22] used interval tree data structure to locate both (i) the maximum- and (ii) the minimum-point enclosing rectangle of a given dimension over a set of points. Although both works provide theoretically optimal bound, they are not suitable for large spatial databases, and a scalable external-memory algorithm – optimal in terms of the I/O complexity – was proposed in [5] (also addressing $(1-\epsilon)$-*approximate MaxRS* and *All-MaxRS* problems). More recently, the problem of indexing spatial objects for efficient MaxRS processing was addressed in [35]. In this work, we used the method of [22] to recompute MaxRS only at certain KDS events, however, we proposed pruning strategies to reduce the number of such invocations. We note that an indexing scheme based on a static sub-division of the 2D plane (cf. [5, 35]) need not to be a good approach for spatio-temporal data because the densities in the spatial partitions will vary over time, and we plan to investigate the problem of efficient indexing techniques for Co-MaxRS as part of our future work.

In [24], an algorithm to process MaxRS queries when the locations of the objects are bounded by an underlying road network is presented. Complementary to this, in [4] the solution is proposed for the *rotating-MaxRS* problem, i.e., allowing non axis-parallel rectangles. Recently, [1] proposed methods to monitor MaxRS queries in spatial data streams – objects appear or disappear dynamically, but do not change their locations. Although [1], [4], and [24] deal with interesting variants of the traditional MaxRS problem, they do not consider the settings of mobile objects.

In this work, we relied on the KDS framework, introduced and practically evaluated in [2]. The KDS-like data structure was used to process critical events at which the current MaxRS solution may change. To estimate the quality of a KDS, [2] considered performance measures such as the time-complexity of processing KDS events and computing certificate failure times, the size of KDS, and bounds on the maximum number of events associated with an object. We used the same measures to evaluate the quality of our approach.

Circular (Co-)MaxRS: A special note is in order for the, so called, circular MaxRS [3] – which is, the region R is a disk instead of a rectangle. Arguably, this problem is $\Theta(n^2)$ and one of the main reasons is that the combinatorial complexity of the boundary of the intersection of a set of disks is not constant (unlike axes-parallel rectangles). This, in turn, would increase the $n \log n$ factor in our algorithms to n^2 – and the continuous variant of the circular MaxRS implies maintaining intersections of sheared cylinders instead of sheared boxes. We also note that this case (counting variant) bears resemblance to works that have tackled problems in trajectory clustering [13]. More specifically, [8] introduced the concept of flocks as a group of trajectories who are moving together within a given disk and for a given time, and [12] introduced the (less constrained) concept of trajectory convoys. These works, while similar in spirit to a continuous variant of the circular MaxRS – have not explicitly addressed the problem of detecting (and maintaining) the disk which contains the maximum number of moving objects, nor have considered weights of the objects. We re-iterate that the results in [8] show that some of the proposed algorithms have complexities similar in magnitude to the worst-case complexity of the Co-MaxRS. An approximate solution to the static variant of the circular MaxRS was presented in [5] (approximating the disk with the minimum bounding square) and our current Co-MaxRS solution can be readily applied towards the approximated variant.

7. CONCLUSION AND FUTURE WORKS

We addressed the problem of determining the locations of a given axes-parallel rectangle R so that the maximum number of moving objects from a given set of trajectories is inside R. In contrast to the MaxRS problem first studied by the computational geometry community [11, 22], the Continuous MaxRS (Co-MaxRS) solution may change over time. To avoid checking the validity of the answer-set at every clock-tick, we identified the critical times at which the answer to Co-MaxRS may need to be re-evaluated, corresponding to – events occurring when the dual rectangles of the moving objects change their topological relationship. To speed up the processing of Co-MaxRS we used the kinetic data structures (KDS) paradigm and proposed two pruning heuristics: (1) eliminating events from KDS; and (2) eliminating the objects not affecting the answer (when re-computation of Co-MaxRS is necessary). While our algorithms mostly focused on the moving objects (resp. rectangles) defining the answer set, the possible volume(s) (in terms of 2D space + time) swept by the Co-MaxRS can be straightforwardly derived. Our experiments, over both real and synthetic data sets, showcased that the proposed heuristics enabled significant speed-ups in terms of the overall computation time from the upper bound on the time complexity.

There are numerous extensions of our work. One task is to devise a suitable indexing structure that will minimize the I/O overheads when trajectories data sets need to reside on a secondary storage or even on cloud [6], and to investigate the trade-offs between processing time vs. approximate answer to Co-MaxRS [5]. While, intuitively, our approaches seem "transferable" to the case of circular Co-MaxRS, we still need to have a more thorough investigation of the pruning effects in the KDS – and a related challenge is to investigate Co-MaxRS when the rectangles are in general positions (i.e., not restricted to be axes-parallel) [4]. In our solution there may be cases where Co-MaxRS has discontinuities – i.e., the current MaxRS needs to instantaneously change its location. Clearly, in practice one may want to have a realistic time-budget for the MaxRS to "travel" from one such location to another – which is another challenge to be addressed, in terms of lost precision. Other natural extensions of this setting are to investigate the k-variant of Co-MaxRS – i.e., the case of multiple mobile cameras jointly guaranteeing a continuous maximal coverage, as well as the effective management of Co-MaxRS for real time location updates.

8. REFERENCES

[1] D. Amagata and T. Hara. Monitoring MaxRS in spatial data streams. In *19th International Conference on Extending Database Technology, EDBT*, 2016.

[2] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1), 1999.

[3] B. M. Chazelle and D.-T. Lee. On a circle placement problem. *Computing*, 36(1-2), 1986.

[4] Z. Chen, Y. Liu, R. C.-W. Wong, J. Xiong, X. Cheng, and P. Chen. Rotating MaxRS queries. *Information Sciences*, 305, 2015.

[5] D. W. Choi, C. W. Chung, and Y. Tao. Maximizing Range Sum in external memory. *ACM Trans. Database Syst.*, 39(3):21:1–21:44, Oct. 2014.

[6] A. Eldawy and M. F. Mokbel. SpatialHadoop: A MapReduce framework for spatial data. In *31st IEEE International Conference on Data Engineering, ICDE*, 2015.

[7] K. Feng, G. Cong, S. S. Bhowmick, W. C. Peng, and C. Miao. Towards best region search for data exploration. In *ACM SIGMOD International Conference on Management of Data*, pages 1055–1070. ACM, 2016.

[8] J. Gudmundsson and M. J. van Kreveld. Computing longest duration flocks in trajectory data. In *ACM GIS Conference*, 2006.

[9] R. H. Güting and M. Schneider. *Moving objects databases.* Elsevier, 2005.

[10] F. J. Harvey and K. J. Krizek. Commuter bicyclist behavior and facility disruption. Technical report, 2007.

[11] H. Imai and T. Asano. Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. *Journal of algorithms*, 4(4), 1983.

[12] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen. Discovery of convoys in trajectory databases. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1), 2008.

[13] P. Kalnis, N. Mamoulis, and S. Bakiras. On discovering moving clusters in spatio-temporal data. 2005.

[14] S. Ke, J. Gong, S. Li, Q. Zhu, X. Liu, and Y. Zhang. A hybrid spatio-temporal data indexing method for trajectory databases. *Sensors*, 14(7), 2014.

[15] M. Koubarakis, T. Sellis, A. Frank, S. Grumbach, R. Güting, C. Jensen, N. Lorentzos, Y. Manolopoulos, E. Nardelli, B. Pernici, H.-J. Scheck, M. Scholl, B. Theodoulidis, and N. Tryfona, editors. *Spatio-Temporal Databases – the CHOROCHRONOS Approach.* Springer-Verlag, 2003.

[16] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data: The next frontier for innovation, competition, and productivity, 2011.

[17] M. Mas-ud Hussain, A. Wang, and G. Trajcevski. Co-MaxRS: Continuous Maximizing Range-Sum query. Technical Report NU-EECS-16-08, Dept. of EECS, Northwestern University, 2016.

[18] M. F. Mokbel, L. Alarabi, J. Bao, A. Eldawy, A. Magdy, M. Sarwat, E. Waytas, and S. Yackel. MNTG: An extensible web-based traffic generator. In *Advances in Spatial and Temporal Databases.* 2013.

[19] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal access methods. *IEEE Data Eng. Bull.*, 26(2), 2003.

[20] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *ACM SIGMOD*, 2004.

[21] Y. Nakayama, D. Amagata, and T. Hara. An efficient method for identifying MaxRS location in mobile ad hoc networks. In *Database and Expert Systems Applications - 27th International Conference, DEXA*, 2016.

[22] S. C. Nandy and B. B. Bhattacharya. A unified algorithm for finding maximum and minimum object enclosing rectangles and cuboids. *Computers & Mathematics with Applications*, 29(8), 1995.

[23] N. Pelekis and Y. Theodoridis. *Mobility Data Management and Exploration.* Springer, 2014.

[24] T.-K. Phan, H. Jung, and U.-M. Kim. An efficient algorithm for Maximizing Range Sum queries in a road network. *The Scientific World Journal*, 2014, 2014.

[25] J. B. Rocha-Junior, A. Vlachou, C. Doulkeridis, and K. Nørvåg. Efficient processing of top-k spatial preference queries. *Proceedings of the VLDB Endowment (PVLDB)*, 4(2), 2010.

[26] L. Sha, P. Lucey, Y. Yue, P. Carr, C. Rohlf, and I. A. Matthews. Chalkboarding: A new spatiotemporal query paradigm for sports play retrieval. In *21st International Conference on Intelligent User Interfaces, IUI*, 2016.

[27] S. Shekhar and S. Chawla. *Spatial databases: A tour*, volume 2003. Prentice Hall Upper Saddle River, NJ, 2003.

[28] Y. Tao, X. Hu, D. Choi, and C. Chung. Approximate MaxRS in spatial databases. *Proceedings of the VLDB Endowment (PVLDB)*, 6(13), 2013.

[29] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *International Conference on Very Large Data Bases (VLDB)*, 2003.

[30] C. R. Vicente, D. Freni, C. Bettini, and C. S. Jensen. Location-related privacy in geo-social networks. *IEEE Internet Computing*, 15(3), 2011.

[31] D. Wu, N. Mamoulis, and J. Shi. Clustering in geo-social networks. *IEEE Data Eng. Bull.*, 38(2), 2015.

[32] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *IEEE International Conference on Data Engineering (ICDE)*, 2005.

[33] Y. Zheng. Trajectory data mining: An overview. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(3), 2015.

[34] Y. Zheng, L. Zhang, X. Xie, and W. Y. Ma. Mining interesting locations and travel sequences from GPS trajectories. In *ACM International Conference on World Wide Web.* ACM, 2009.

[35] X. Zhou, W. Wang, and J. Xu. General purpose index-based method for efficient MaxRS query. In *27th International Conference - DEXA*, 2016.

Exact and Approximate Algorithms for Finding k-Shortest Paths with Limited Overlap

Theodoros Chondrogiannis
Free University of Bozen-Bolzano
tchond@inf.unibz.it

Panagiotis Bouros
Aarhus University
pbour@cs.au.dk

Johann Gamper
Free University of Bozen-Bolzano
gamper@inf.unibz.it

Ulf Leser
Humboldt-Universität zu Berlin
leser@informatik.hu-berlin.de

ABSTRACT

Shortest path computation is a fundamental problem in road networks with various applications in research and industry. However, returning only the shortest path is often not satisfying. Users might also be interested in alternative paths that are slightly longer but have other desired properties, e.g., less frequent traffic congestion.

In this paper, we study alternative routing and, in particular, the k-Shortest Paths with Limited Overlap (k-SPwLO) query, which aims at computing paths that are (a) sufficiently dissimilar to each other, and (b) as short as possible. First, we propose MultiPass, an exact algorithm which traverses the network $k-1$ times and employs two pruning criteria to reduce the number of paths that have to be examined. To achieve better performance and scalability, we also propose two approximate algorithms that trade accuracy for efficiency. OnePass$^+$ employs the same pruning criteria as MultiPass, but traverses the network only once. Therefore, some paths might be lost that otherwise would be part of the solution. ESX computes alternative paths by incrementally removing edges from the road network and running shortest path queries on the updated network. An extensive experimental analysis on real road networks shows that: (a) MultiPass outperforms state-of-the-art exact algorithms for computing k-SPwLO queries, (b) OnePass$^+$ runs significantly faster than MultiPass and its result is close to the exact solution, and (c) ESX is faster than OnePass$^+$ (though slightly less accurate) and scales for large road networks and large values of k.

CCS Concepts

•Information systems → Geographic information systems; Database query processing;

Keywords

Alternative Routing;Road Networks;Query Services

1. INTRODUCTION

Computing the shortest path between two locations in a road network is a fundamental problem that has attracted lots of attention by both the research community and the industry. Traditionally, the shortest path problem is addressed by Dijkstra's algorithm [9]. Additionally, a plethora of pre-processing based methods have been proposed that answer shortest path queries in almost constant time, even for continental sized networks [7, 11, 21, 25].

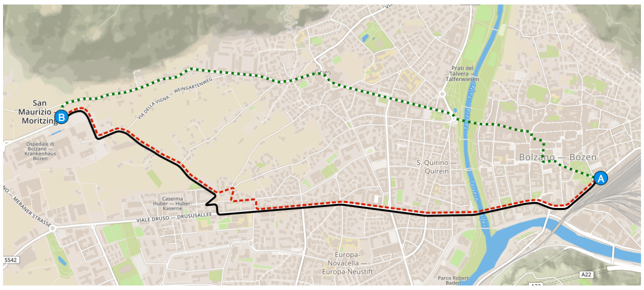

San Maurizio — Moritzing
B
Ospedale di Bolzano — Krankenhaus Bozen
VIA DELLA VIGNA — WEINGARTENWEG
Gries
Prati del Talvera — Talferwiesen
Talvera — Talfer
Bolzano — Bozen
A
S. Quirino — Quirein
Caserma Huber — Huber-Kaserne
VIALE DRUSO — DRUSUSALLEE
MERANER STRASSE
SS42
Europa-Novacella — Europa-Neustift
Parco Robert Baden
A22

Figure 1: Motivational example

However, in many real-world scenarios, determining solely the shortest path is not enough. Most commercial route planning applications and navigation systems offer alternatives that might be longer than the shortest path but have other desirable properties (e.g., lower fuel consumption), leaving the final decision to the user. Alternative routes are also very useful for the transportation of goods using a fleet of vehicles, i.e., transportation of humanitarian aid through unsafe regions. By distributing the load into vehicles that follow different routes, the probability that at least some of the goods will arrive at the destination safely can be increased. Another interesting scenario arises in emergency situations such as natural disasters and terrorist attacks. To avoid panic and potential catastrophic collisions while dealing with the aftermath of such events, evacuation plans should include, apart from the shortest, alternative paths that are sufficiently dissimilar to each other.

A first take on providing alternative routes with no prior information is to solve the K-shortest paths problem [10, 15, 24]. In most cases, though, the returned paths share large stretches, and therefore, they are of little practical value to the user. Consider the scenario illustrated in Figure 1, which shows three different paths from location A to B representing the central train station and the hospital in the city of Bolzano, respectively. The solid/black line indicates the shortest path from A to B while the dashed/red line indicates the next path by length; notice how similar these two paths are. On the other hand, the green/dotted line indicates a third path, which is clearly longer than the other two but significantly different from the shortest path. In practice, the paths cover very distant

Series ISSN: 2367-2005

10.5441/002/edbt.2017.37

parts of the city's road network. In many application scenarios, the green/dotted path would be considered as a better alternative to the shortest path, compared to the dashed/red path.

Existing literature has approached alternative routing from different perspectives. Notable works include methods which aim at computing alternative routes either by incrementally building a set of dissimilar paths [12] or by employing edge penalties [2]. The proposed methods, though, typically give no guarantees regarding the length of the alternative paths. Other approaches [1, 3, 5] first generate a large number of candidates and then, in a post-processing step, consider a number of constraints and criteria in order to determine the final alternative paths. However, in these works alternatives are defined based solely on their individual similarity to the shortest path, which results in alternative paths that are very similar to each other and, hence, of limited interest to the user.

Contributions. In this paper, we focus on the problem of finding *k-Shortest Paths with Limited Overlap* (k-SPwLO), previously introduced in [6]. A k-SPwLO query aims at computing paths that are (a) *sufficiently dissimilar to each other (based on a user-specified similarity threshold)*, and (b) *as short as possible*. In [6], we presented the OnePass algorithm for processing k-SPwLO queries. The algorithm outperforms a baseline solution which enumerates paths in increasing length order, but, in reality, OnePass is not practical even for mid-sized road networks. To this end, we propose MultiPass, an exact algorithm which extends and improves OnePass by employing an additional pruning criterion. In contrast to OnePass, which traverses the road network once and expands only those paths that qualify the similarity constraint, MultiPass traverses the network $k-1$ times, but examines and expands only the most *promising* paths. Any path that cannot lead to a solution is pruned. Our experimental analysis shows that MultiPass always outperforms OnePass, and, in most cases, by a large margin.

Despite its significant performance advantage over OnePass, also MultiPass cannot scale in practice for large road networks, a fact that is backed by our extensive experimental evaluation. In this spirit, we propose two approximate methods that trade result quality for efficiency. Our first approximate algorithm, OnePass$^+$, employs the pruning power of MultiPass, but traverses the road network only once, similar to OnePass. Thereby, OnePass$^+$ may prune some partial paths that, in a subsequent iteration, could become part of an alternative path. Our second approximate algorithm, ESX, computes alternative paths by incrementally removing edges from the road network that belong to previously recommended paths, and running shortest path queries on the updated network. Essentially, ESX reduces the search for alternative paths to a set of shortest path queries which require much less time to be processed. In our extensive experimental evaluation, we show that the approximate algorithm OnePass$^+$ is significantly faster than the exact algorithm MultiPass, while recommending alternative paths that are almost as short as the alternatives in the exact k-SPwLO set. We also show that ESX is the fastest algorithm and is scalable even for large road networks (i.e., one million nodes) and large values of k.

Outline. The rest of the paper is organized as follows. Section 2 briefly discusses the related work on providing alternative paths. In Section 3, we formally define the k-SPwLO problem and revisit our evaluation methodology from [6]. In Section 4, we present MultiPass, a novel exact algorithm for processing k-SPwLO queries, and conduct a preliminary experimental analysis comparing MultiPass to OnePass. Next in Section 5, we investigate the approximate evaluation of k-SPwLO. We first discuss a baseline method SVP$^+$, based on existing literature and then propose our approximate algorithms OnePass$^+$ and ESX. The results of our detailed experimental evaluation are reported in Section 6. Finally, Section 7 concludes the paper and points to future work.

2. RELATED WORK

A common approach for alternative routing is to first compute a large set of candidate paths, then examine the candidate paths with respect to a number of constraints (e.g., their length or the nodes they cross) and determine the final result set. In [5], the authors build two shortest path trees, one from the source and one from the target, and then look for paths that appear in both trees simultaneously, termed *plateaus*. This approach was revisited and formally defined in [3] (where the concept of alternative graphs has been introduced with the same functionality as the plateaus) and further improved in [17]. Abraham et al. [1] introduced the notion of *single-via paths*. The method runs Dijkstra's algorithm two times, once from the source s and once from the target t while reversing the edges of the road network. Then, for each node n apart from s and t, the algorithm constructs a single-via path by concatenating the shortest path from s to n and the shortest path from n to t. The algorithm evaluates each (simple) single-via path by employing a set of user-defined constraints, i.e., length, local optimality and stretch, and rejects all single-via paths that violate these constraints. Compared to our k-SPwLO problem, none of the aforementioned methods tackles the problem of computing multiple alternative paths that are dissimilar to each other; in contrast, the similarity only to the shortest path is considered.

Penalty-based methods generate a set of paths that are dissimilar to the shortest path by adding a penalty on the weights of the edges of the shortest path. For example, Akgun et. al. [2] propose a method which doubles the weight of each edge that lies on the shortest path. The alternative paths are computed by repeatedly running Dijkstra's algorithm on the input road network, each time with the updated weights. A similar approach is adopted in [14], where the penalty is computed in terms of both the path overlap and the total turning cost, i.e., how many times the user would have to switch between roads when following a path. The main shortcoming of penalty-based methods is that there is no intuition behind the value of the penalty applied before each subsequent iteration. In general, using a large penalty value would result in diverse but possibly very long alternative paths. On the other hand, using a small penalty value would require the algorithm to perform more iterations in order to find the desired result. Even so, penalty-based methods cannot provide a formal result set. Our last approximate algorithm ESX can also be viewed as a penalty based method where the penalty added to the weight of selected edges is $+\infty$.

To the best of our knowledge, the problem tackled in [12] is the most similar to our k-SPwLO problem. The authors devise a solution which extends Yen's algorithm [24] to produce paths that qualify a similarity constraint. In particular, given a source s and a target t, starting from the shortest path, the algorithm produces a set of candidate paths by modifying the previously found path. Among the candidate paths, the algorithm chooses the one that is most dissimilar to the previously found path and continues until a sufficiently dissimilar path is found. Apparently, the algorithm does not examine paths in length order but only based on their similarity. Thus, it does not compute alternative paths that are as short as possible, but only dissimilar. Naturally, a user finds more value in paths that are also as short as possible.

Xie et. al. [23] define alternative shortest paths using edge avoidance. Given the shortest path $p(s \rightarrow t)$ and an edge e on p, the alternative path is the shortest path from s to t which avoids edge e. To compute alternative paths, they build upon the concept of distance oracles [20] and distance sensitivity oracles [4] and propose *iSPQF*, a quadtree-based spatial data structure inspired by [19]. Compared to our work, iSQPF computes only one alternative path instead of a set. Moreover, the use-case is different as Xie et al. find alternative paths by explicitly avoiding forbidden edges, while k-SPwLO considers the similarity between paths to propose alternative paths.

Finally, the task of alternative routing can be based on the pareto-optimal paths for multi-criteria networks [8, 13, 16]. A path p is part of the pareto-optimal set (or the route skyline) P if p is not dominated by another path $p' \in P$. Hence, path p dominates p' iff p is not worse than p' in all criteria/dimensions of the network (e.g., distance, travel time, gas consumption) and strictly better than p' in at least one of those criteria. Our definition of alternative routing, i.e., the k-SPwLO query, is not a multi-criteria optimization problem; the paths recommended by k-SPwLO cannot be obtained by first computing the pareto-optimal path set.

3. BACKGROUND

Let $G{=}(N, E)$ be a *directed weighted graph* representing a *road network* with set of nodes N and set of edges $E \subseteq N{\times}N$. The nodes of G represent road intersections and the edges represent road segments. Each edge $(n_x, n_y) \in E$ has an assigned positive weight w_{xy}, which captures the cost of moving from node n_x to n_y, e.g., travel time or distance. A (simple) *path* $p(s \rightarrow t)$ from a source node s to a target node t (or just p, if s and t are clear from the context) is a connected and cycle-free sequence of edges $\langle (s, n_x), \ldots, (n_y, t)\rangle$. The length $\ell(p)$ of a path p equals the sum of the weights of all contained edges. The *shortest path* between two nodes, $p_0(s{\rightarrow}t)$, is the path that has the shortest length among all paths connecting s to t. The length of the shortest path is also termed the *(network) distance* between s and t, i.e., $d(s, t){=}\ell(p_0(s{\rightarrow}t))$.

Problem Definition. In [6], we introduced the problem of *k-Shortest Paths with Limited Overlap* (k-SPwLO) in order to recommend alternative paths a user may take to reach her destination. In particular, let P be a set of paths from a node s to another node t on a road network G. A path $p'(s{\rightarrow}t)$ is called *alternative* to set P if p' is sufficiently dissimilar to every path $p \in P$. More formally, the similarity of p' to p is determined by their overlap ratio:

$$Sim(p', p){=}\frac{\sum_{(n_x,n_y)\in p'\cap p} w_{xy}}{\ell(p)}, \tag{1}$$

where $p' \cap p$ denotes the set of edges shared by p' and p. Then, given a similarity threshold θ, path p' is alternative to set P *iff* $Sim(p', p) \leq \theta, \forall p \in P$ holds.

Now, given a source node s and a target node t, a k-SPwLO (s, t, θ, k) query returns a set of k paths from s to t, sorted in increasing length order, such that:

(a) the shortest path $p_0(s{\rightarrow}t)$ is always included,

(b) all k paths are pairwise dissimilar with respect to the similarity threshold θ, and

(c) all k paths are as short as possible.

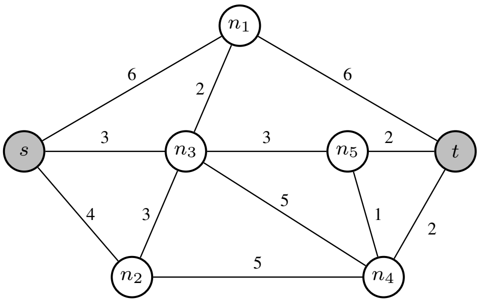


Figure 2: Running example.

Consider the road network in Figure 2. The shortest path from s to t is $p_0{=}\langle (s, n_3), (n_3, n_5), (n_5, t)\rangle$ with length $\ell(p_0){=}8$. Assume that P contains only the shortest path, i.e., $P{=}\{p_0\}$ and consider paths $p_1{=}\langle (s, n_3), (n_3, n_5), (n_5, n_4), (n_4, t)\rangle$ and $p_2{=}\langle (s, n_3), (n_3, n_4), (n_4, t)\rangle$ with $\ell(p_1){=}9$ and $\ell(p_2){=}10$, respectively, as alternatives to P. Path p_1 shares edges (s, n_3) and (n_3, n_5) with p_0, which gives $Sim(p_1, p_0){=}(w_{s,3} + w_{3,5})/\ell(p_0) = 6/8 = 0.75$, whereas $Sim(p_2, p_0){=}w_{s,3}/\ell(p_0) = 3/8 = 0.38$. Assuming a similarity threshold $\theta{=}0.5$, only p_2 is alternative to P.

Note that the asymmetric similarity metric of Equation 1 allows us to exclude needlessly long paths. Following up on our previous example, consider the shortest path p_0 and the paths $p_3{=}\langle (s, n_3), (n_3, n_4), (n_4, t)\rangle$ and $p_4{=}\langle (s, n_3), (n_3, n_2), (n_2, n_4), (n_4, t)\rangle$ with $\ell(p_3){=}10$ and $\ell(p_4){=}13$, respectively. The use of a symmetric similarity metric such as the Jaccard distance would indicate that p_4 is less similar to p_0 than p_3, although the shared length of both p_3 and p_4 with p_0 is the same. With the asymmetric definition of Equation 1 we avoid such cases. Furthermore, the pairwise dissimilarity is guaranteed as long as $\ell(p') \geq \ell(p)$ (proof excluded due to lack of space).

Evaluating k-SPwLO. A naïve approach for evaluating k-SPwLO queries is to iterate over all paths connecting s to t and compute their pairwise similarity. Naturally, such a solution is not practical. A potential improvement is to examine paths in increasing order of their length, which allows us not to examine all possible paths $p(s \rightarrow t)$. This idea was captured by the baseline method in [6], but the computation cost is still prohibitively high.

To further reduce the search space, we first introduced a pruning criterion in [6] based on the following simple observation. Let $p(s{\rightarrow}n)$ be a path connecting source s to a node n, and $p_i(s{\rightarrow}t) \in P_{LO}$ be an already recommended path. Assume that p is extended to reach target t, resulting in path $p'(s{\rightarrow}t)$. As p' contains all edges shared by p and p_i, its similarity to p_i is at least equal to the similarity of path p, i.e., $Sim(p', p_i) \geq Sim(p, p_i)$. Hence, given a threshold θ, if there exists $p_i \in P_{LO}$ such that $Sim(p, p_i) \geq \theta$, path p can be safely discarded. This observation is formally captured by the following lemma:

LEMMA 1. *Let P_{LO} be the set of already recommended paths. If p is an alternative path to P_{LO} with respect to a threshold θ,then $Sim(p', p_i) \leq \theta$ holds for every subpath p' of p and all $p_i \in P_{LO}$.*

We used this Lemma 1 as pruning criterion in the OnePass algorithm. The algorithm traverses the road network, expanding every path from the source node s that satisfies Lemma 1. OnePass employs a min priority queue in order to examine paths in increasing order of their length. Each time a new path is recommended, i.e., added to the result set P_{LO}, an update procedure takes place for all remaining incomplete paths $p(s \rightarrow n)$ in the priority queue. The algorithm terminates when either k paths are added to the result set or all paths from s to t qualifying Lemma 1 are examined.

OnePass can be viewed as an extension of *Fox's algorithm* [10] for computing the K-shortest paths. Fox's algorithm traverses the road network expanding every path from source node s. At each iteration, the algorithm expands up to K nodes, allowing each node to be expanded up to K times. It terminates when the target node has been expanded K times. The time complexity of Fox's algorithm is $\mathcal{O}(|E| + K \cdot |N| \cdot \log |N|)$. In contrast to Fox's algorithm, OnePass allows each node to be visited an unlimited number of times. Each node can be visited by OnePass as many times as the number of paths from s to t. Note that enumerating all paths from s to t is a $\#P$-complete problem [22]. OnePass terminates when either k paths are recommended or all paths from s to t qualifying Lemma 1 are examined. Hence, the complexity of OnePass is $\mathcal{O}(|E| + K \cdot |N| \cdot \log |N|)$, where K is the number of shortest paths that have to be computed in order to cover the k results of the k-SPwLO query.

4. AN EFFICIENT EXACT ALGORITHM

Despite employing the pruning criterion of Lemma 1, OnePass still has to expand and examine a large portion of all possible $p(s{\rightarrow}t)$ paths. In this section, we propose a novel label-setting algorithm termed MultiPass to enhance the computation of k-SPwLO. The algorithm employs an additional powerful pruning criterion which significantly reduces the search space by avoiding expanding *non-promising* paths. Our experimental analysis in Section 4.3 demonstrates the advantage of MultiPass over OnePass in practice using real-world road networks.

4.1 Pruning Non-Promising Paths

Let $p_0(s{\rightarrow}t)$ be the shortest path from a source node s to a target node t as illustrated in Figure 3. In addition, let $p_i(s{\rightarrow}n)$ and $p_j(s{\rightarrow}n)$ be two distinct paths from source s to a node n of the shortest path p_0 such that $\ell(p_i){<}\ell(p_j)$. Assuming that both p_i and p_j are extended to reach target t following the same path $p(n{\rightarrow}t)$, then any extension of p_i will be shorter than the respective extension of p_j. Furthermore, let $Sim(p_i, p_0){\leq}Sim(p_j, p_0)$, i.e., the overlap ratio of p_i with p_0 is equal or lower than the ratio of p_j with p_0. Due to the monotonicity of the similarity function (Equation (1)), any extension of p_i to n will have the same or less overlap ratio with p_0 compared to the respective extension of p_j. In other words, for any extension of p_j there will always be a shorter extension of p_i with less or equal overlap ratio with p_0, and therefore, p_j can be pruned.

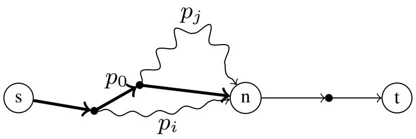


Figure 3: Pruning paths with Lemma 2.

The same idea can be utilized to prune the search space when computing the shortest alternative path to a set of paths P. Consider again p_i,p_j with $\ell(p_i){<}\ell(p_j)$ and $Sim(p_i, p_0){\leq}Sim(p_j, p_0)$. Path p_j is pruned if for every path $p \in P$ the overlap ratio $Sim(p_i, p)$ is lower or equal to $Sim(p_j, p)$. This pruning criterion is formally captured by the following lemma:

LEMMA 2. *Let P be a set of paths from a source node s to a target node t, and p_i, p_j be two paths from source s to some node n. If $\ell(p_i){<}\ell(p_j)$ and $\forall p \in P : Sim(p_i, p){\leq}Sim(p_j, p)$ hold, then path p_j cannot be part of the shortest alternative path to P, and we write $p_i \prec_P p_j$.*

PROOF. We prove the lemma by contradiction. Assume that an extension $p'_j{=}\langle(s, *), \ldots, (*, n), \ldots, (*, t)\rangle$ of $p_j(s{\rightarrow}n)$ to target t is the shortest alternative path to P. Then, we show that an extension $p'_i{=}\langle(s, *), \ldots, (*, n), \ldots, (*, t)\rangle$ of $p_i(s{\rightarrow}n)$ to target t is also an alternative path and it will be examined and recommended before p'_j.

According to the definition of alternative path, $Sim(p'_j, p){\leq}\theta$ holds $\forall p \in P$ and following Lemma 1 $Sim(p_j, p){\leq}\theta$ also holds $\forall p \in P$. Furthermore, due to the $\forall p \in P_{LO} : Sim(p_i, p){\leq}Sim(p_j, p)$ assumption of Lemma 2, we get that $Sim(p_i, p){\leq}\theta$ holds $\forall p \in P$.

As extension paths p'_i and p'_j share the same sequence of edges connecting n to target t, we deduce that (a) $Sim(p'_i, p){\leq}\theta$ holds $\forall p \in P$, i.e., p'_i is alternative to P and (b) $\ell(p'_i){<}\ell(p'_j)$ which means that p'_i will be examined before p'_j. □

The pruning criterion of Lemma 2 can be utilized to compute the shortest alternative to a set of paths as follows. Let P be the set of paths for which we want to compute the shortest alternative path, and P_n be the set of paths from s to a node n created during the expansion of all paths from s. If set P_n contains a path $p'(s{\rightarrow}n)$ such that (a) p' is longer than any path $p_n \in P_n \setminus \{p'\}$ and (b) for every path $p \in P$ the overlap ratio $Sim(p', p)$ is higher than the ratio $Sim(p_n, p)$ for all paths $p_n \in P_n \setminus \{p'\}$, then p' can be pruned. Note that the addition of a path in P_n may render condition (b) not applicable for another path already contained in P_n. To ensure that the set P_n contains only paths for which both (a) and (b) hold, every time a new path is added to P_n, we have to check whether condition (b) still holds for all paths in the set.

Unfortunately, Lemma 2 cannot be employed directly for the computation of k-SPwLO queries. Consider again the example in Figure 3. Let p_0 be the only path in the set of currently recommended alternative paths P. If during the search for the next alternative p_1 to P, p_j is pruned because $p_i \prec_P p_j$ holds, p_j cannot be part of the shortest alternative to P. However, there is no guarantee that p_j will not be part of the shortest alternative to both p_0 and p_1. In particular, if p_i is part of p_1, then, during the search for the next alternative to $P{=}\{p_0, p_1\}$, p_i might be pruned much earlier by Lemma 1. Hence, we have to compute k-SPwLO queries in an iterative way. Each time a new alternative is added to the k-SPwLO result set, we have to re-start the search for the next alternative from the beginning.

4.2 The MultiPass Algorithm

Next, we present MultiPass, which employs both pruning criteria of Lemma 1 and Lemma 2 to enhance the computation of k-SPwLO queries. The algorithm has the following key features. For each node n of the road network, MultiPass maintains a set of labels $\Lambda(n)$. Each label represents a path from s to n and is of the form $\langle n, p(s{\rightarrow}n)\rangle$[1]. MultiPass traverses the road network k-1 times. At each iteration, the algorithm examines paths from s in increasing order of their length and expands every path $p(s{\rightarrow}n)$ from s to a node n for which the following holds: (a) its similarity with any already computed result does not exceed the input threshold θ (Lemma 1) and (b) its extension can possibly lead to the shortest alternative path during the current iteration (Lemma 2). Every time a new path $p_n(s{\rightarrow}n)$ that qualifies conditions (a) and (b) is found, a label $\langle n, p_n\rangle$ is added to $\Lambda(n)$, and MultiPass removes all paths from $\Lambda(n)$ which do not qualify condition (b). As soon as a path to target t is found, MultiPass terminates current round, discards all stored labels, and re-traverses the network from source s. The algorithm terminates after k paths are added to result set P_{LO} or

[1] In practice, MultiPass stores only the predecessor of each label during the expansion. By tracing backwards each step of the expansion, the actual path can be retrieved at any time.

ALGORITHM 1: MultiPass

```
Input: Road network $G(N,E)$, source node $s$, target node $t$,
       # of results $k$, similarity threshold $\theta$
Output: Set $P_{LO}$ of $k$ paths
1  $P_{LO} \leftarrow$ {shortest path $p_0(s \rightarrow t)$};
2  while $|P_{LO}| < k$ and last round updated $P_{LO}$ do
3      initialize min-priority queue $\mathcal{Q}$ with $\langle s, \emptyset \rangle$;
4      $\forall n \in N : \Lambda(n) \leftarrow \emptyset$;
5      while $\mathcal{Q}$ not empty do
6          $\langle n, p_n \rangle \leftarrow \mathcal{Q}.pop()$;            ▷ Current path
7          if $n{=}t$ then
8              $P_{LO} \leftarrow P_{LO} \cup \{p_n\}$; ▷ Update result set
9              break;
10         else
11             foreach outgoing edge $(n, n_c) \in E$ do
12                 $p_c \leftarrow p_n \circ (n, n_c)$;    ▷ Expand path $p_c$
13                 if $\exists p_i \in P_{LO} : Sim(p_c, p_i) \geq \theta$ then
14                     continue;    ▷ Pruned by Lemma 1
15                 else if $\exists \langle n_c, p'_c \rangle \in \Lambda(n_c) : p'_c \prec_{P_{LO}} p_c$ then
16                     continue;    ▷ Pruned by Lemma 2
17                 else
18                     remove from $\mathcal{Q}$ and $\Lambda(n_c)$ all
                       $\langle n_c, p'_c \rangle : p_c \prec_{P_{LO}} p'_c$;      ▷ Lemma 2
19                     $\mathcal{Q}.push(\langle n_c, p_c \rangle)$;
20                     $\Lambda(n_c) \leftarrow \Lambda(n_c) \cup \{\langle n_c, p_c \rangle\}$;
21 return $P_{LO}$;
```

the last iteration failed to find an alternative path. In the latter case, a complete set of k-SPwLO with respect to given θ and k values cannot be computed.

Algorithm 1 illustrates the pseudocode of MultiPass. The algorithm initializes P_{LO} with the shortest path $p_0(s \rightarrow t)$ (Line 1) and employs a min priority queue $\mathcal{Q}$ to traverse the road network. Before each traversal round, $\mathcal{Q}$ is initialized to $\langle s, \emptyset \rangle$ (Line 3) and the algorithm associates each node n with a (initially empty) set of labels $\Lambda(n)$ (Line 4). At each round in between Lines 5 and 20, MultiPass first pops label $\langle n, p_n \rangle$ for current path p_n in Line 6. If n is the target t, then p_n is added to P_{LO} and the round terminates (Lines 7–9). Otherwise, the algorithm expands the current path p_n considering all outgoing edges (n, n_c) (Lines 10-16). For each new path $p_c \leftarrow p_n \circ (n, n_c)$ (Line 12), the algorithm checks whether p_c qualifies the pruning criteria of Lemma 1 (Lines 13-14) and Lemma 2 (Lines 15-16). If the new path p_c qualifies both pruning criteria, MultiPass removes from $\mathcal{Q}$ and $\Lambda(n_c)$ every label representing a path p'_n such that $p_c \prec_{P_{LO}} p'_n$ (Line 19). Finally, MultiPass adds the new label to $\mathcal{Q}$ (Line 19) and $\Lambda(n_c)$ (Line 20) and proceeds with popping the next label from $\mathcal{Q}$.

To achieve an efficient implementation, for each label $\langle n, p_n \rangle$ MultiPass also stores a vector V_{Sim} containing the overlap ratio of p_n with all paths that were in P_{LO} at the time when the label was created. Due to the monotonicity of Equation 1, the overlap ratios stored in V_{Sim} can be updated incrementally. When a new label is created and added to $\mathcal{Q}$, our implementation of MultiPass performs lazy updates for $\mathcal{Q}$ and retains a black list to ignore labels representing pruned paths after they are removed from $\mathcal{Q}$. In order to consider results in P_{LO} that are added after the creation of the label, each time a label is popped MultiPass compares the size of V_{Sim} stored in the popped label to $|P_{LO}|$ and, if necessary, computes the missing overlaps and updates V_{Sim}.

EXAMPLE 1. *We demonstrate MultiPass using the road network of Figure 4 and the k-SPwLO$(s, t, 0.5, 3)$ query. During initialization, the shortest path* $p_0(s \rightarrow n) = \langle (s, n_3), (n_3, n_5), (n_5, t) \rangle$ *is computed and added to* P_{LO}.

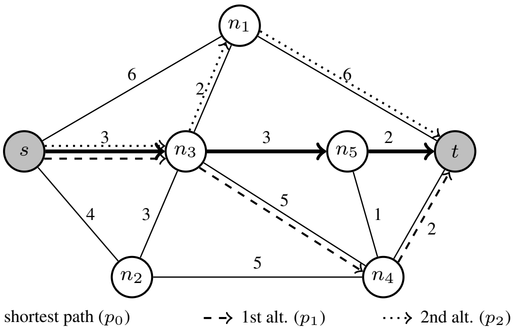


Figure 4: Result of k-SPwLO (s,t,0.5,3) query.

Starting from s, the first path examined by MultiPass is $p(s \rightarrow n_3) = \langle (s, n_3) \rangle$. *The overlap ratio* $Sim(p(s, n_3), p_0) = 3/8 = 0.375$ *is below the similarity threshold* $\theta = 0.5$*; hence* $p(s, n_3)$ *is not pruned. The same holds for the next two paths examined, which are* $p(s \rightarrow n_2) = \langle (s, n_2) \rangle$ *and* $p(s \rightarrow n_1) = \langle (s, n_1) \rangle$.

Next, MultiPass examines paths $p(s \rightarrow n_5) = \langle (s, n_3), (n_3, n_5) \rangle$, $p'(s \rightarrow n_1) = \langle (s, n_3), (n_3, n_1) \rangle$, $p'(s \rightarrow n_2) = \langle (s, n_3), (n_3, n_2) \rangle$ *and* $p(s \rightarrow n_4) = \langle (s, n_3), (n_3, n_4) \rangle$. *For path* $p(s \rightarrow n_5)$ *the overlap ratio* $Sim(p(s, n_5), p_0) = 6/8 = 0.75$ *exceeds the similarity threshold of* 0.5 *and so, path* $p(s \rightarrow n_5)$ *is pruned (Lemma 1). For path* $p'(s \rightarrow n_1)$ *the overlap ratio* $Sim(p'(s, n_1), p_0) = 0.375$ *does not exceed the similarity threshold. Since node* n_1 *has already been visited by path* $p(s \rightarrow n_1)$*, we check Lemma 2. We have* $Sim(p'(s \rightarrow n_1), p_0) > Sim(p(s \rightarrow n_1), p_0)$ *and for the length* $\ell(p'(s \rightarrow n_1)) < \ell(p(s, n_1))$*. Therefore, Lemma 2 cannot be applied and path* $p'(s \rightarrow n_1)$ *is not pruned. On the contrary, for path* $p'(s \rightarrow n_2)$ *we have* $Sim(p'(s \rightarrow n_2), p_0) > Sim(p(s \rightarrow n_2), p_0)$ *and* $\ell(p'(s \rightarrow n_2)) > \ell(p(s \rightarrow n_2))$. *In this case, the criterion of Lemma 2 is applied and path* $p'(s \rightarrow n_2)$ *is pruned. Finally, for path* $p(s \rightarrow n_4)$ *the overlap ratio* $Sim(p(s \rightarrow n_4), p_0) = 0.375$ *does not exceed the similarity threshold, hence, the path is not pruned.*

MultiPass continues the execution of the current round until the alternative path $p_1(s \rightarrow t) = \langle (s, n_3), (n_3, n_4), (n_4, t) \rangle$ *with* $\ell(p_1) = 10$ *is found and subsequently added to* P_{LO}. *Next, MultiPass performs the second round in the same fashion, computes the alternative path* $p_2(s \rightarrow t) = \langle (s, n_3), (n_3, n_1), (n_1, t) \rangle$ *and completes the result set* P_{LO}.

Compared to OnePass, MultiPass traverses the road network $k{-}1$ times instead of once (hence, the name of the algorithm). Each round works independently, i.e., builds a new path tree by expanding all paths that qualify both pruning criteria. As a result, at each round, MultiPass may potentially re-expand and re-examine paths already processed in previous rounds. On the other hand, by employing Lemma 2, the number of paths that MultiPass has to examine (including the paths examined multiple times) is lower than the number of paths processed by OnePass. Finally, OnePass has to check the simplicity of every new path, i.e., whether any cycles are contained, while MultiPass does not need to perform such a check, as Lemma 2 ensures that all non-simple paths are pruned.

Complexity analysis. Given a k-SPwLO query, MultiPass first computes the shortest path $p_0(s \rightarrow t)$ from source node s to target t. Naturally, the cost of this step is independent of the number of requested paths k and the similarity threshold θ. For the computation of $p_0(s \rightarrow t)$, any shortest path algorithm can be employed, e.g., Dijkstra's algorithm [9] which requires $\mathcal{O}(|E| + |N| \cdot \log |N|)$.

To find each subsequent alternative path, MultiPass expands all paths from source s that qualify the pruning criterion of Lemma 1. However, as the value of the similarity threshold θ approaches 1, the number of paths pruned by Lemma 1 significantly drops, which means that MultiPass returns the K-Shortest paths. Furthermore in practice, there exists no formula for estimating the number of paths pruned by the pruning criterion of Lemma 2. Hence, each round of MultiPass becomes equivalent to Fox's algorithm [10] with a complexity $\mathcal{O}(|E| + K \cdot |N| \cdot \log |N|)$, where K is the number of the shortest paths that have to be computed to cover the k-SPwLO result. Since MultiPass has to perform $k-1$ rounds to compute a k-SPwLO query, its total runtime complexity is $\mathcal{O}(k(|E|+K \cdot |N| \cdot \log |N|))$ where $K \gg k$. We have shown in [6] that the number of K-shortest paths that need to be computed in order to cover the k-SPwLO set is very high; in extreme cases, MultiPass may have to construct all paths from s to t.

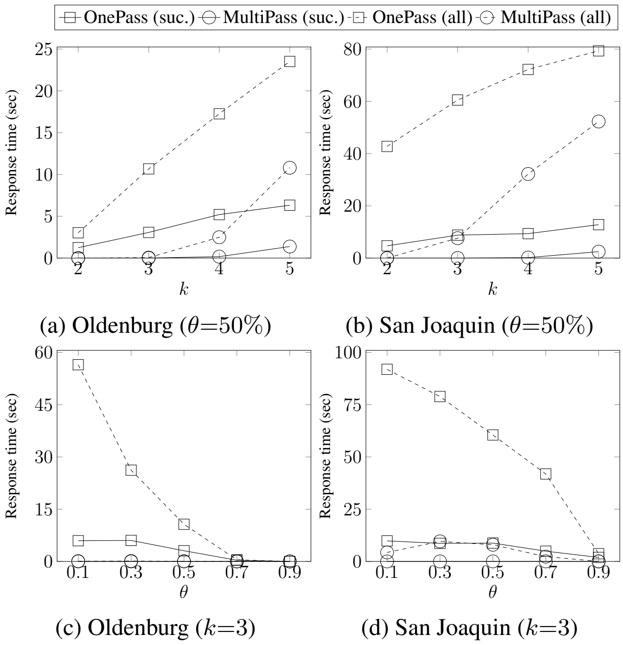
Finally, note that the time complexity of MultiPass is worse than the time complexity of OnePass. However, we show in our experimental evaluation that MultiPass is much faster than OnePass. The reason for this inconsistency is that, although by employing the pruning criterion of Lemma 2 MultiPass examines much less paths than OnePass, there can be no formal guarantees for the number of paths that are pruned. Although MultiPass has worse theoretical time complexity, in practice it is much more efficient than OnePass.

Optimization. As discussed in [6] for OnePass, the performance of MultiPass can be further enhanced by employing a lower bound, $\underline{d}(n,t)$, for the network distance $d(n,t)$ of every node n to the target t. By employing such a lower bound, MultiPass traverses at each round the network in an A^*-like fashion and directs the search towards the target, which avoids visiting nodes that are far away. In order to derive tight $\underline{d}(n,t)$ lower bounds, we first reverse the edges of the road network and then run Dijkstra's algorithm from target t to every node n of the network [18]. In practice, at the beginning of the MultiPass execution, instead of simply computing the shortest path from s to t, we compute the shortest path from target t to each node n in the road network.

4.3 Experimental Evaluation

To demonstrate its efficiency, we compare MultiPass against OnePass presented in [6] using real road networks. For each algorithm, we measure the average response time and the number of examined labels (i.e., paths) over 1,000 random queries varying parameters k and θ. Due to the high execution time of OnePass, our experiments involve only the road networks for the city of Oldenburg (6,105 nodes and 14,058 edges) and the city of San Joaquin (18,263 nodes and 47,594 edges). We also consider a timeout of 120 seconds for the evaluation of each query.

Figure 5 reports the response times of MultiPass and OnePass. The continuous lines show the time for the queries for which both algorithms finished their execution in less than 120 seconds, whereas the dashed lines show the time for all 1,000 queries including those which did not finish within 120 seconds. In Figures 5a and 5b, we observe that the performance of both OnePass and MultiPass deteriorates as the number k of requested paths increases. For all values of k though, MultiPass is clearly faster than OnePass, and in most cases MultiPass is at least two times faster. Another interesting observation is that the performance curve of OnePass is almost linear, i.e., each iteration requires approximately the same time. For instance, for Oldenburg (Figure 5a) the algorithm needs similar time to find the third and the fourth alternative path. On the other hand, MultiPass needs more time for each subsequent result. This behavior can be explained by the fact that MultiPass restarts and re-expands paths. With regard to parameter θ, we observe in Figures 5c and 5d that in all cases, MultiPass is faster than OnePass. Especially for the lowest values of θ, i.e., 0,1 and 0.3, MultiPass outperforms OnePass by at least an order of magnitude. The performance of OnePass is close to MultiPass only for θ=0.9, where the computed paths can be very similar.



(a) Oldenburg (θ=50%)

(b) San Joaquin (θ=50%)

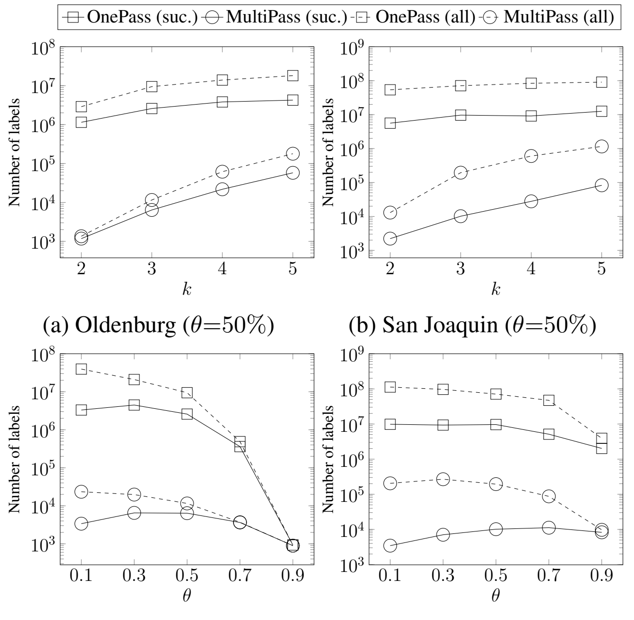
(c) Oldenburg (k=3)

(d) San Joaquin (k=3)

Figure 5: Performance comparison of MultiPass and OnePass varying requested paths k and similarity threshold θ.

To provide a better insight on the performance of MultiPass and OnePass, we report in Figure 6 the number of labels/paths each algorithm needs to examine before returning the k-SPwLO result. We observe that in all scenarios MultiPass examines significantly fewer paths than OnePass (even though we count the total number of paths from all rounds of MultiPass, hence some paths may be counted more than once). With respect to the similarity threshold θ, we observe the following important trade-off. As θ increases, the pruning power of Lemma 1 deteriorates, and both OnePass and MultiPass construct more paths (supporting measurements are not included due to lack of space). However at the same time, the next result can be determined earlier and, hence, the total runtime drops. In addition, as θ decreases, the pruning power of Lemma 2 increases, and more partial paths can be pruned. This explains the behavior of MultiPass, where the number of examined paths initially increases, but after θ=0.5 it goes down.

Finally, in Table 1 we report the percentage of timed-out/failed queries for timeout values of 30, 60 and 120 seconds. First, we observe that the failure rate of OnePass is, in most cases, much higher than the failure rate of MultiPass. More specifically, for the road network of Oldenburg, the failure rate of OnePass is more than 10% when k>3 or θ<50%. For the road network of San Joaquin, apart from the case where k=3 and θ=90%, the failure rate of OnePass is more than 30%, even when the timeout is set to 120 seconds. On the contrary, the timeout rate of MultiPass for the road network of Oldenburg is in all cases below 10%. For the road network of San Joaquin, the failure rate of MultiPass is below 10%,



(a) Oldenburg (θ=50%)   (b) San Joaquin (θ=50%)

(c) Oldenburg (k=3)   (d) San Joaquin (k=3)

Figure 6: Comparison of examined paths by OnePass and MultiPass varying requested paths k and similarity threshold θ.

except for the cases where k>3. However, even in cases where the failure rate of MultiPass is the highest (θ=50% and k>3), it is still significantly lower than the failure rate of OnePass.

road network	θ	k	OnePass			MultiPass		
			30	60	120	30	60	120
Oldenburg	0.1	3	46.9	45.4	44.5	0	0	0
	0.3	3	22.8	20.5	17.8	0	0	0
	0.5	3	9.1	7.7	6.6	0	0	0
	0.7	3	0.4	0.1	0.1	0	0	0
	0.9	3	0	0	0	0	0	0
	0.5	2	2.7	0.2	1.5	0	0	0
	0.5	4	15.2	12.8	10.7	2.6	1.4	0.7
	0.5	5	20.4	17.5	15	9.6	7.4	6.2
San Joaquin	0.1	3	77.5	76	74.6	3.2	1.8	1.3
	0.3	3	66.8	65.2	63.2	8.1	5.6	4.4
	0.5	3	52.3	49.4	46.6	6.8	5.1	3.5
	0.7	3	35.8	33.6	32.1	2.1	0.9	0.3
	0.9	3	3.1	2.3	1.6	0	0	0
	0.5	2	36.5	34.1	33.2	0	0	0
	0.5	4	61	59.3	56.8	28.9	25.5	22.5
	0.5	5	67.5	64.8	62.3	45.2	42	39.1

Table 1: Failure rate (%) for timeout set to 30, 60 and 120 sec.

5. APPROXIMATE ALGORITHMS

Our experimental analysis in Section 4.3 showed that MultiPass clearly outperforms OnePass presented in [6]. However, despite employing Lemma 2, MultiPass still has to examine a large number of paths, which essentially renders the algorithm not applicable to large-scale road networks. In view of this, we next investigate the approximate evaluation of k-SPwLO queries. In particular, we first discuss a baseline method, termed SVP$^+$, which builds on top of existing literature, and then we propose two novel approximate algorithms, termed OnePass$^+$ and ESX.

5.1 A Baseline Solution

Our baseline algorithm, denoted by SVP$^+$, builds upon the notion of *single-via paths*, which was introduced as an alternative routing technique in [1]. As we discussed in Section 2, the original method considers the similarity of single-via paths only with regard to the shortest path and disregards the pairwise dissimilarity of all results. Instead of employing the objective criteria as in [1], SVP$^+$ iterates over the set of single-via paths aiming to find a subset of k paths which are (a) sufficiently dissimilar to each other and (b) as short as possible. Intuitively, the main idea behind SVP$^+$ is similar to the baseline method for computing k-SPwLO queries discussed in Section 3. However, instead of iterating over all possible paths connecting a source node s to a target node t and computing their pairwise overlap ratio, SVP$^+$ iterates over the much smaller set of single-via paths. As the results of k-SPwLO are not necessarily singe-via paths, SVP$^+$ can only provide approximate answers to the queries.

ALGORITHM 2: SVP$^+$

```
Input: Road network $G(N,E)$, source node $s$, target node $t$,
       # of results $k$, similarity threshold $\theta$
Output: Set $P_{LO}$ of $k$ paths
1  initialize min-priority queue $\mathcal{Q}$ with $\emptyset$;
2  $T_{s\rightarrow N} \leftarrow$ shortest path tree from $s$ to all $n \in N$;
3  $T_{N\rightarrow t} \leftarrow$ shortest path tree from all $n \in N$ to $t$;
4  foreach $n \in N$ do
5      $\mathcal{Q}.push(\langle n, d(s,n)+d(n,t)\rangle)$;
6  $P_{LO} \leftarrow \{$shortest path $p_0(s \rightarrow t)\}$;
7  while $P_{LO}$ contains less than k paths and $\mathcal{Q}$ not empty do
8      $\langle n, \ell(p_n) \leftarrow \mathcal{Q}.pop()$;
9      $p_n \leftarrow$ RetrieveSingleViaPath$(T_{s\rightarrow N}, T_{N\rightarrow t}, n)$;
10     if $Sim(p_n, p) \leq \theta$ for all $p \in P_{LO}$ then
11         add $p_n$ to $P_{LO}$;            ▷ Update result set
12 return $P_{LO}$;
```

Algorithm 2 illustrates the pseudocode of SVP$^+$. First, the algorithm computes the two shortest path trees, one from s to every node of G (Line 2) and one from every node of G to t (Line 3). During this step all distances $d(s,n)$ and $d(n,t)$ are also computed. The algorithm orders the nodes based on the sum of $d(s,n)+d(n,t)$, which is also the length of the single-via path of n, using a min priority queue $\mathcal{Q}$ (Lines 4-5). In Line 6, the result set P_{LO} is initialized with p_0, i.e., the shortest path from s to t. Note that the shortest path p_0 is actually the shortest single-via path and, hence, no additional computation is required. At each iteration between lines 7 and 11, SVP$^+$ pops from the queue the top element representing a node n (Line 8) and retrieves the single-via path p_n for node n (Line 9). Then, SVP$^+$ checks in Line 10 whether p_n is sufficiently dissimilar to all paths currently in P_{LO}; if so, p_n is added to P_{LO} (Line 11). The algorithm terminates and returns the P_{LO} set when either k paths have been added to P_{LO} or there exist no more single-via paths to examine, i.e., queue $\mathcal{Q}$ is depleted.

5.2 Approximate OnePass

We propose next a novel approximate algorithm, denoted by OnePass$^+$, which combines the feature of OnePass to scan the graph only once with the pruning power of Lemma 2. OnePass$^+$ has the following key features. Given a source node s and a target node t, OnePass$^+$ traverses the road network expanding every path $p(s\rightarrow n)$ from source s to a node n that qualifies both Lemma 1 and Lemma 2. This procedure is the same with each distinct round of MultiPass. In contrast to MultiPass though, each time a new result is added to the result set P_{LO}, an update procedure takes place for all remaining incomplete paths $p(s\rightarrow n)$. In particular, for every incomplete path $p(s\rightarrow n)$, OnePass$^+$ computes the overlap of p with the newly found result and, then, p is checked against Lemma 1 with respect to the updated P_{LO}. The same update procedure is also employed by OnePass. The algorithm terminates when either k paths are recommended or all paths from s to t qualifying Lemma 1 and Lemma 2 have been examined.

By not restarting after the computation of each new result, OnePass$^+$ avoids expanding the network multiple times. However, the fact that OnePass$^+$ does not restart the expansion after each round implies that the next best path might get pruned and, hence, OnePass$^+$ cannot guarantee that the exact solution will be found. We already explained in Sec. 4 that for the MultiPass algorithm to find the exact solution, the restart is required as a path that is pruned as non-promising during the current round, may be promising during the next round. All such paths are excluded permanently from the search space of OnePass$^+$. Nevertheless, this case applies to only a small subset of the paths from a source s to a target t and, hence, the average length of paths in the P_{LO} set is expected to be close to the optimal one.

Algorithm 3 illustrates the pseudocode of OnePass$^+$. The algorithm employs a min priority queue $\mathcal{Q}$ (initialized with source s) to traverse the road network. Result set P_{LO} is initialized with p_0, i.e., the shortest path from s to t (Line 1). In between Lines 4 and 21, OnePass$^+$ examines the contents of $\mathcal{Q}$ until either k paths are found or $\mathcal{Q}$ is depleted. At each iteration, a label $\langle n, p_n \rangle$ is popped from $\mathcal{Q}$ (Line 5). If node n is the target t, then p_n is added to P_{LO} (Line 7) and the same update procedure as in OnePass takes place (Lines 8-10), i.e., all paths p_h with $Sim(p_h, p_c) > \theta$ are discarded. Otherwise, the algorithm expands the current path p_n considering all outgoing edges (n, n_c) (Lines 12-21). OnePass checks whether the new path $p_c \leftarrow p_n \circ (n, n_c)$ qualifies the pruning criteria of both Lemma 1 (Lines 14-15) and Lemma 2 (Lines 16-17) and updates $\mathcal{Q}$ and $\Lambda(n_c)$ accordingly. Finally, OnePass adds a new label for p_c to $\mathcal{Q}$ (Line 20) and $\Lambda(n_c)$ (Line 21) and proceeds with popping the next label from $\mathcal{Q}$.

Similar to MultiPass, for each label our implementation of OnePass$^+$ maintains and updates incrementally a vector V_{Sim} containing the overlaps of p_n with all paths that where in P_{LO} at the time when the label was created. Furthermore, OnePass$^+$ also performs lazy updates for $\mathcal{Q}$. That is, for labels that have already been created and added to $\mathcal{Q}$, OnePass$^+$ updates V_{Sim} only at the time when a label is popped from $\mathcal{Q}$. OnePass$^+$ also retains a black list to ignore labels representing pruned paths.

ALGORITHM 3: OnePass$^+$

```
   Input: Road network $G(N, E)$, source node $s$, target node $t$,
          # of results $k$, similarity threshold $\theta$
   Output: Set $P_{LO}$ of $k$ paths
1  $P_{LO} \leftarrow \{$shortest path $p_0(s \to t)\}$;
2  initialize min-priority queue $\mathcal{Q}$ with $\langle s, \emptyset \rangle$;
3  $\forall n \in N : \Lambda(n) \leftarrow \emptyset$;
4  while $P_{LO}$ contains less than k paths and $\mathcal{Q}$ not empty do
5      $\langle n, p_n \rangle \leftarrow \mathcal{Q}.pop()$;                ▷ Current path
6      if $n = t$ then
7          $P_{LO} \leftarrow P_{LO} \cup \{p_n\}$;      ▷ Update result set
8          foreach label $\langle n', \ell(p_{n'}) \rangle$ in $\mathcal{Q}$ do
9              if $Sim(p_{n'}, p_i) > \theta, \forall p_i \in P_{LO}$ then
10                 remove $\langle n', \ell(p_{n'}) \rangle$ from $\mathcal{Q}$ ;    ▷ Lemma 1
11     else
12         foreach outgoing edge $(n, n_c) \in E$ do
13             $p_c \leftarrow p_n \circ (n, n_c)$;          ▷ Expand path $p_c$
14             if $\exists p_i \in P_{LO} : Sim(p_c, p_i) \geq \theta$ then
15                 continue;          ▷ Pruned by Lemma 1
16             else if $\exists \langle n_c, p'_c \rangle \in \Lambda(n_c) : p'_c \prec_{P_{LO}} p_c$ then
17                 continue;          ▷ Pruned by Lemma 2
18             else
19                 remove from $\mathcal{Q}$ and $\Lambda(n_c)$ all
                   $\langle n_c, p'_c \rangle : p_c \prec_{P_{LO}} p'_c$;            ▷ Lemma 2
20                 $\mathcal{Q}.push(\langle n_c, p_c \rangle)$;
21                 $\Lambda(n_c) \leftarrow \Lambda(n_c) \cup \{\langle n_c, p_c \rangle\}$;
22 return $P_{LO}$;
```

Complexity Analysis. Similar to MultiPass, given a k-SPwLO query from a node s to a node t, OnePass$^+$ first computes $p_0(s \to t)$ using any shortest path algorithm, e.g., Dijkstra, and adds it to the result set. To compute alternatives, OnePass$^+$ traverses the road network expanding every path $p(s \to n)$ from source s to a node n that qualifies both Lemma 1 and Lemma 2. As we discussed in the cost analysis of MultiPass, there can be no formal guarantees regarding the number of paths that are pruned using either pruning criterion. In the worst case when no paths are pruned, OnePass$^+$ is equivalent to OnePass and Fox's algorithm. Therefore, the time complexity of OnePass$^+$ is also $\mathcal{O}(|E| + K \cdot |N| \cdot \log |N|)$, where K is the number of shortest paths that have to be computed in order to cover the k-SPwLO result.

5.3 Edge Subset Exclusion

Finally, we present our second approximate algorithm, denoted by ESX, which computes k-SPwLO by iteratively excluding edges from the road network. The idea behind ESX is the following. Given a road network G, a source node s and a target node t, the algorithm first adds the shortest path p_0 to the result set P_{LO}, similar to all previously described methods. Next, ESX removes an edge of p_0 from the road network and computes the shortest path p_c from s to t on the updated road network[2]. If the overlap of path p_c with p_0 does not violate the similarity threshold θ, then p_c is added to the result set P_{LO}. Otherwise, the algorithm proceeds with removing more edges from the road network. If P_{LO} contains more than one paths, ESX removes an edge from path $p \in P_{LO}$ for which the similarity $Sim(p_c, p)$ is the highest. At each iteration, ESX removes only one edge from some path in P_{LO}. The process is repeated until a path that does not violate the similarity threshold θ is found. To compute more alternatives, the algorithm continues by removing more edges until another alternative is found, or until there are no more edges to remove.

Removing an edge from the road network may cause the network to become disconnected and prevent any subsequent iteration from finding a valid path. To avoid such a case, the algorithm has to make sure that any edge affecting the connectivity of the road network is never removed. To this end, after removing an edge from the road network, if the shortest path search fails to find a path connecting s and t, then ESX re-inserts the edge in the road network and marks it as *non-removable*. Edges marked as non-removable cannot be removed at any iteration.

The order in which we remove the edges from the road network affects both the quality of the result and the performance of ESX. However, determining the optimal order is prohibitively expensive. Therefore, to determine which edge to remove at each iteration, we employ a heuristic based on the following observation: the more shortest paths cross an edge, the greater the probability that the removal of this edge will cause a detour and lead the next result faster. As it is also prohibitively expensive to compute the all-pairs

[2] In practice, the edges are not actually deleted from the road network but only marked as such in order to be ignored by the search.

shortest paths, ESX performs a local check. Given an edge $e(a, b)$ on some path $p \in P_{LO}$, let $E_{inc}(a)$ be the set of all incoming edges $e(n_i, a)$ to a from some node $n_i \in N \backslash \{b\}$ and $E_{out}(b)$ be the set of all outgoing edges $e(b, n_j)$ from b to some node $n_j \in N \backslash \{a\}$. First, ESX computes the set P_s which contains the shortest paths from every node $n_i \in E_{inc}(a)$ to every node $n_j \in E_{out}(b)$. Then, ESX defines the set P'_s which contains all paths $p \in P'_s$ that cross edge e. Finally, ESX assigns a priority to edge e, denoted by $prio(e)$, which is set to $|P'_s|$.

EXAMPLE 2. *Consider our running example in Figure 7, where $p_0(s \rightarrow t) = \langle (s, n_3), (n_3, n_5), (n_5, t) \rangle$ is the shortest path from s to t and the only path currently in P_{LO}. For edge (n_3, n_5) we compute the shortest path from every node in $\{s, n_1, n_2, n_4\}$ to every node in $\{n_4, t\}$. Three shortest paths, $p(n_1 \rightarrow n_4)$, $p(s \rightarrow n_4)$ and $p(s \rightarrow t)$, cross edge (n_3, n_5) (bold lines). On the other hand, the rest of the shortest paths, e.g., shortest path $p(n_2 \rightarrow t)$ (dashed line), do not cross edge (n_3, n_5). Therefore, the priority of edge (n_3, n_5) is $prio(n_3, n_5) = 3$. In the same fashion, we compute the priorities for edges (s, n_3) and (n_3, t), and we have $prio(s, n_3) = 0$ and $prio(n_5, t) = 0$.*

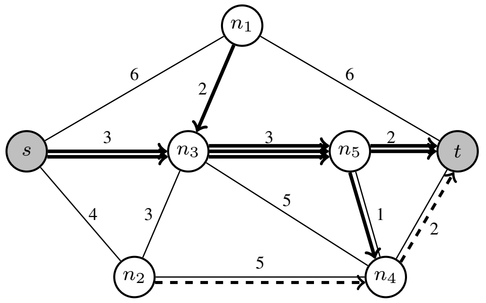


Figure 7: Computing the priority of edge (n_3, n_5).

Algorithm 4 illustrates the pseudocode of ESX. First, the algorithm initializes P_{LO} with the shortest path p_0 in Line 1 and creates a max-heap H_0, associated with p_0, in which all the edges of p_0 are enheaped and sorted based on their priority (Line 2). The algorithm also initializes the set E_{DNR} of non-removable edges (Line 3). ESX enters the outer loop in Line 4 and continues until either k results are found or there are no more edges to be removed from the graph. Next, the algorithm sets p_c to the last result found and enters the inner loop (Line 6). At each iteration the algorithm chooses p_{max} as the path in P_{LO} which has the maximum overlap $Sim(p_c, p_{max})$ and which contains edges in H_{max} (the max-heap associated with p_{max}) that can be removed from the graph. Then, the algorithm deheaps edge e_{tmp} from H_{max} (Line 8) and checks whether e_{tmp} is in E_{DNR}, i.e., it is marked as non-removable (Line 9). If it is not, edge e_{tmp} is removed (Line 10) and the algorithm computes the shortest path p_{tmp} on the updated graph (Line 11). In Lines 12-15 the algorithm checks whether p_{tmp} is a valid path and, if not, re-inserts e_{tmp} to the graph and marks it as non-removable. Otherwise, the algorithm sets p_c to p_{tmp} and proceeds to the next iteration. Finally, when the inner loop is finished, the algorithm checks if p_c is a valid alternative (there is also the possibility that all the heaps are empty and no more edge can be removed). If p_c is valid, it is added to P_{LO} and a new max-heap H_c associated with p_c is initialized with the edges of p_c. Finally, after the outer loop is finished, the algorithm returns P_{LO} in Line 20.

ALGORITHM 4: ESX

```
   Input: Road network $G(N, E)$, source node $s$, target node $t$,
          # of results $k$, similarity threshold $\theta$
   Output: Set $P_{LO}$ of $k$ paths
1  $P_{LO} \leftarrow$ {shortest path $p_0(s \rightarrow t)$};
2  initialize max-heap $H_0 \leftarrow \langle e_i, \texttt{prio}(G, e_i) \rangle, \forall e_i \in p_0$;
   ▷ Every $H_i$ is associated with $p_i$
3  initialize $E_{DNR} \leftarrow \emptyset$;
4  while $P_{LO}$ contains less than k paths and $\exists H_i$ not empty do
5      set $p_c \leftarrow$ last path added to $P_{LO}$;
6      while $max\{Sim(p_c, p_i) : p_i \in P_{LO}$ and $H_i$ not empty
       $\} > \theta$ do
7          Edge $e_{tmp} \leftarrow H_i.pop()$;
8          if $e_{tmp} \in E_{DNR}$ then
9              continue;
10         $G.remove(e_{tmp})$;
11         Path $p_{tmp} \leftarrow \texttt{ShortestPath}(G, s, t)$;
12         if $p_{tmp}$ is null then
13             re-insert $e_{tmp}$ to $G$;
14             insert $e_{tmp}$ to $E_{DNR}$;
15             continue;
16         $p_c \leftarrow p_{tmp}$;
17     if $max\{Sim(p_c, p_i) : p_i \in P_{LO}\}$ then
18         add $p_c$ to $P_{LO}$;
19         initialize max-heap $H_c \leftarrow \langle e_j, \texttt{prio}(G, e_j) \rangle$,
           $\forall e_j \in p_c$;
20 return $P_{LO}$;
```

EXAMPLE 3. *We demonstrate the functionality of ESX using again the road network of Figure 4 and the $k{-}SPwLO(s, t, 0.5, 2)$ query. During initialization, the shortest path $p_0(s \rightarrow t) = \langle (s, n_3), (n_3, n_5), (n_5, t) \rangle$ is computed and added to the result set P_{LO}. First, we compute the priority of each edge of the shortest path. Having computed the priorities, we first remove edge (n_3, n_5), which is the edge with the highest priority. Then, we compute the shortest path $p'(s \rightarrow t)$ on the updated graph. The shortest path is $p'(s \rightarrow t) = \langle (s, n_3), (n_3, n_4), (n_4, t) \rangle$ with $\ell(p'(s \rightarrow t)) = 10$. We check the overlap of the new path with the original shortest path and find that $Sim(p'(s \rightarrow t), p_0) = 0.375$, which does not exceed the similarity threshold. Therefore, $p'(s \rightarrow t)$ is added to the P_{LO} set.*

Complexity Analysis. ESX reduces the search for an alternative path to a set of ordinary shortest path queries. In particular, given a road network $G = (N, E)$ let P_{LO} be the result set of a k-SPwLO(s, t, θ, k) query containing k paths. ESX requires $|P| \times$ total number of edges in P executions of shortest path queries, i.e., the number of shortest path queries that have to be processed is linear to the number k of paths and the size of the result paths. Furthermore, in our implementation of ESX we employed the optimization using lower bounds described in Section 4.2, which reduces the cost for retrieving each shortest path to a minimum; thus, the performance of ESX is significantly optimized.

6. EXPERIMENTAL EVALUATION

In this section, we report the results of an experimental evaluation of the algorithms for processing k-SPwLO queries. We use seven different road networks shown in Table 2. To assess the runtime performance, we measure the average response time over 1,000 random queries (i.e., pairs of nodes), varying the number k of requested paths and the similarity threshold θ. In each experiment, we vary one of the two parameters and fix the other to its default value: 3 for k and 0.5 for θ. We also report experiments on

Table 2: Road networks.

road network	# nodes	# edges
Oldenburg	6,105	14,058
San Joaquin	18,263	47,594
Vienna	19,826	54,918
Denver	73,166	196,630
San Francisco	174,956	443,604
New York City	264,346	730,100
Colorado	435,666	1,057,066

the quality of the results computed by the approximate solutions. Given the number k of requested paths, we measure (a) the number of paths returned by each approximate algorithm and (b) the average length of the computed paths in comparison to the length of the shortest path. All algorithms were implemented in C++ and the tests run on a machine with 4 Intel Xeon X5550 (2.67GHz) processors and 48GB main memory running Ubuntu Linux.

6.1 Performance

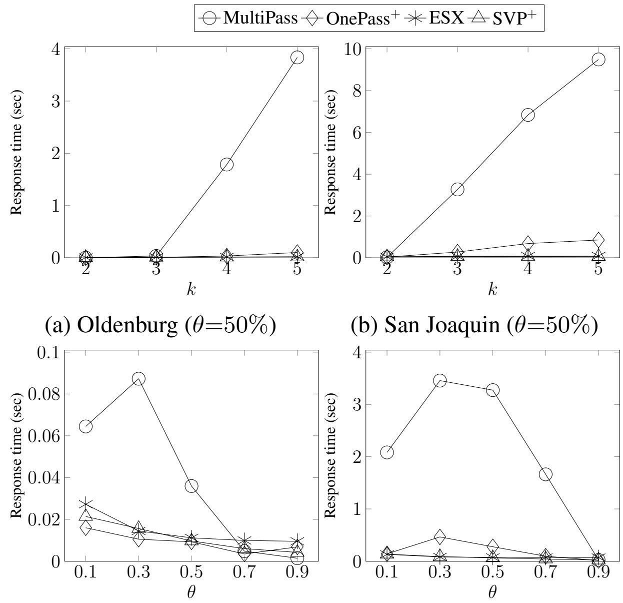
Similar to Section 4.3, we consider a timeout of 120 seconds for each query. The ratio of timed-out/failed queries for OnePass$^+$ was below 10% in all experiments. For SVP$^+$ and ESX, all queries were executed within 120 seconds. Due to space limitations, the performance results shown in this section consider only those queries which were successfully completed by all algorithms.

The first experiment in Figure 8 compares the exact algorithm MultiPass with the approximate solutions SVP$^+$, OnePass$^+$ and ESX. In Figures 8a–b we observe that the runtime of all algorithms increases with the number k of requested paths. As expected, the runtime of the approximate solutions increases only slightly, whereas the exact solution MultiPass deteriorates for large values of k. For $k{>}3$, MultiPass becomes one order of magnitude slower than OnePass$^+$ and more than two orders of magnitude slower than ESX and SVP$^+$. With regard to parameter θ, Figures 8c–d show that for $\theta{<}70\%$, MultiPass is one order of magnitude slower than OnePass$^+$ and two orders of magnitude slower than SVP$^+$ and ESX (for $\theta{=}30\%$). For large values of θ, the performance of MultiPass gets closer to the performance of the approximate algorithms (for $\theta{=}90\%$ MultiPass is even faster than ESX and SVP$^+$).

The next experiment in Figure 9 compares the approximate algorithms using larger datasets. In Figures 9a–b, we vary the parameter k. For small values of k, the difference is not much, while for increasing values of k both SVP$^+$ and ESX clearly outperform OnePass$^+$ up two one order of magnitude. In Figures 9c–d, where the value of θ varies, we observe that OnePass$^+$ is very fast for extreme values of θ ($\theta{=}10\%$ and $\theta{=}90\%$), but it is rather slow for values in between. It is clear that OnePass$^+$ is not practical for large road networks and/or values of $k{>}3$.

Another interesting observation in Figures 8c–d and 9c–d is that the performance of MultiPass and OnePass$^+$ show a local maximum for $\theta{=}0.3$, which indicates the following important trade-off. As θ increases, the pruning power of Lemma 1 deteriorates, and both MultiPass and OnePass$^+$ construct more (partial) paths (supporting measurements are excluded due to lack of space). At the same time, the next result will be determined earlier, and hence the total runtime drops. In addition, as θ decreases, the pruning power of Lemma 2 also increases and more partial paths are pruned. This explains the behavior of MultiPass and OnePass$^+$, where the response time initially increases, but after $\theta{=}0.3$ the runtime of both algorithms goes down.

To summarize the observations in Figures 8 and 9, the approximate solutions clearly outperform the exact algorithm MultiPass. Comparing the approximate solutions, we observe that SVP$^+$ and ESX have similar performance and are the clear winners for the datasets, which are of small and medium size. OnePass$^+$ is the slowest approximate solution.



(a) Oldenburg (θ=50%)

(b) San Joaquin (θ=50%)

(c) Oldenburg (k=3)

(d) San Joaquin (k=3)

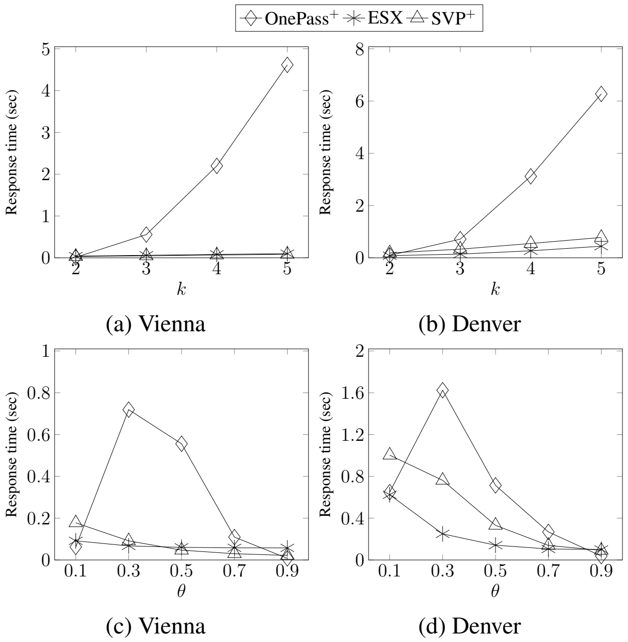
Figure 8: Performance comparison of exact and approximate algorithms varying requested paths k and similarity threshold θ.

6.2 Scalability

From the previous experiments it is clear that both MultiPass and OnePass$^+$ are not scalable. For values of $k{>}2$ both algorithms are prohibitively expensive, even for a mid-sized road network such as Denver. However, the same does not apply for ESX and SVP$^+$. To demonstrate the scalability of ESX and SVP$^+$, we present in Figure 10 the results of an experiment using larger values of $k \in \{4, 8, 12, 16\}$ and we also include larger datasets. We observe that for San Francisco and Colorado, ESX is significantly faster that SVP$^+$ for all values of k. For the road networks of Denver and New York, ESX is faster than SVP$^+$ only for small values of k, whereas SVP$^+$ appears to be faster than ESX for $k{=}12$ and $k{=}16$. The reason for this behavior is that SVP$^+$ computes considerably less alternative paths than ESX (cf. Table 3 and the discussion in Sec. 6.3). Notice that the smaller result set is not due to a timeout, rather the algorithm is not able to find more alternatives. Overall, whenever ESX and SVP$^+$ find approximately the same number of alternative paths, ESX clearly outperforms SVP$^+$.

6.3 Result Quality & Completeness

In Figure 11, we present our experiments that analyze the quality of the computed results. We consider all queries for which each algorithm returned k paths (i.e., no timeout) and compute the average length of the returned paths. Then we compare the average length of each result set to the length of the shortest path. That is, we show how much longer, on average, are the alternative paths with respect to the shortest path. Obviously, the exact results, named k-SPwLO, contain the shortest alternatives. They can be computed by any exact algorithm, such as MultiPass. Looking at the approximate solutions, OnePass$^+$ produces clearly the best alternatives, which are very close to the paths in the exact solution. Both ESX



(a) Vienna

(b) Denver

(c) Vienna

(d) Denver

Figure 9: Performance comparison of approximate algorithms varying requested paths k and similarity threshold θ.

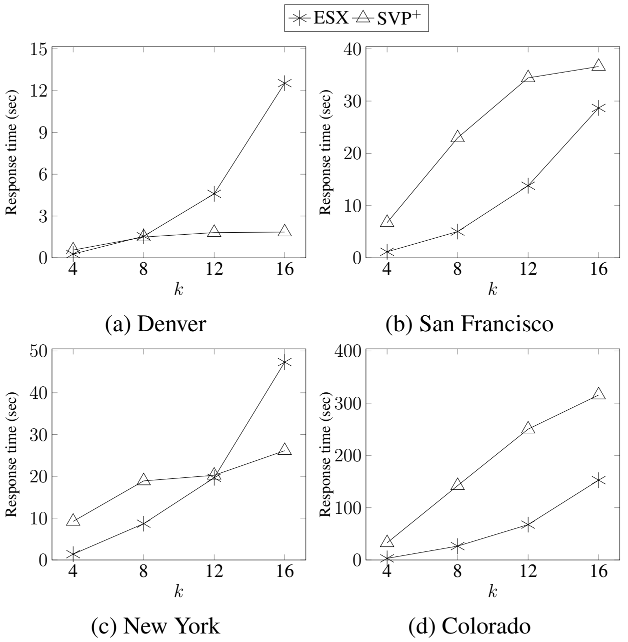
and SVP$^+$ recommend alternatives that, on average, are up to 15% longer than the alternatives in k-SPwLO. The alternatives recommended by ESX, though, are most of the time shorter than the alternatives recommended by SVP$^+$.

The next experiment analyzes the completeness of the result sets. As we have already seen in previous experiments, the algorithms are not always able to compute k alternative paths. Table 3 reports for each algorithm the percentage of queries for which exactly k alternative paths were found. Naturally, the exact solution k-SPwLO has the highest completion ratio. OnePass$^+$ is very close to the exact solution. In particular, for San Joaquin the completion ratio of OnePass$^+$ is always more than 99%. The completion ratio of ESX is lower than OnePass$^+$, but constantly over 95%. Finally, SVP$^+$ has generally the lowest completion ratio (except for k=3, where ESX is slightly worse).

road network	k	k-SPwLO	OnePass$^+$	ESX	SVP$^+$
Oldenburg	2	100	100	100	100
	3	99.9	99.1	98.7	99.5
	4	99.9	98.6	97.1	95
	5	99.9	98.2	95.8	85.6
San Joaquin	2	100	100	100	99.9
	3	100	99.8	98.5	99.5
	4	100	99.7	97.7	97
	5	100	99.3	95.6	94.3

Table 3: Average completeness ratio (%) per query varying requested paths k ($\theta = 50\%$) for all algorithms.

The final experiment in Table 4 compares the quality of ESX and SVP$^+$ by measuring the average number of returned paths for four road networks and values of $k \in \{4, 8, 12, 16\}$. It is evident that ESX returns more alternative paths than SVP$^+$ for all values of k. The number of alternatives returned by ESX is, in all cases, very close to k. In contrast, the number of paths returned by SVP$^+$ is significantly lower than k for k>8. For instance, for New York SVP$^+$ cannot find more than six alternatives per query on average; similar figures can be observed for Denver and San Francisco. Apparently, the set of single-via paths does not contain enough sufficiently dissimilar paths, and hence SVP$^+$ returns more and more incomplete results for an increasing k.



(a) Denver

(b) San Francisco

(c) New York

(d) Colorado

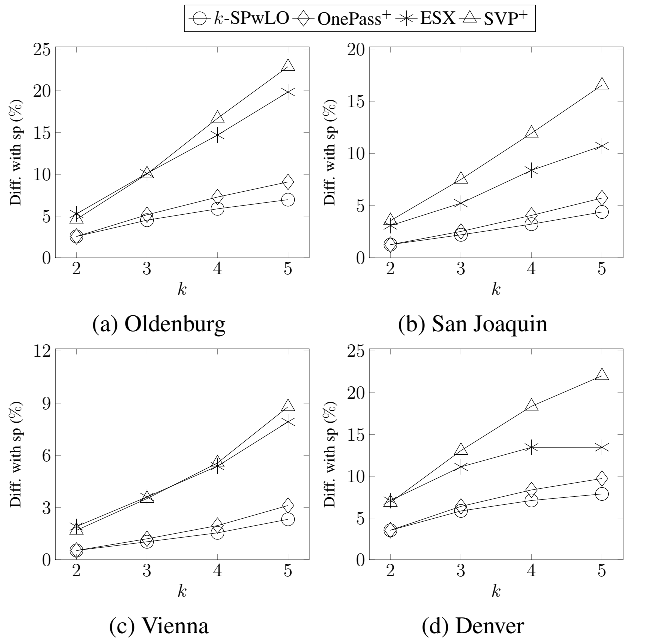
Figure 10: Performance comparison of SVP$^+$ and ESX for $k \in \{2, 4, 8, 16\}$ and $\theta = 50\%$.

road network	k	ESX	SVP$^+$
Denver	4	3.96	3.95
	8	7.72	6.52
	12	11.39	7.14
	16	14.94	7.25
San Francisco	4	3.97	3.95
	8	7.92	7.03
	12	11.81	8.42
	16	15.55	8.80
New York	4	3.97	3.75
	8	7.77	5.49
	12	11.45	5.85
	16	15.02	5.91
Colorado	4	3.97	3.81
	8	7.92	7.87
	12	11.83	10.78
	16	15.71	12.55

Table 4: Average returned results per query varying requested paths k ($\theta = 50\%$) for SVP$^+$ and ESX.

7. CONCLUSIONS

We studied the problem of alternative routing on road networks and, in particular, the efficient computation and approximation of k-SPwLO queries. First, we proposed MultiPass, an exact algorithm which builds upon and optimizes the existing OnePass algorithm by employing one additional pruning criterion. Our experiments showed that MultiPass is the most efficient exact method for evaluating k-SPwLO queries as it outperforms OnePass for nearly every combination of the θ and k parameters and, in most cases, by a large margin. To achieve scalability though, we also introduced two approximate algorithms. OnePass$^+$ employs ideas from both OnePass and MultiPass and achieves to compute a set of dissimilar



(a) Oldenburg

(b) San Joaquin

(c) Vienna

(d) Denver

Figure 11: Result quality varying requested paths k ($\theta = 50\%$).

paths which, in terms of average length, is very close to the exact solution. ESX, our second approximate algorithm, computes alternatives by incrementally removing edges from the road network and running shortest path queries. Through an analytical experimental evaluation we showed that (a) MultiPass is the fastest exact algorithm, outperforming the existing OnePass, (b) OnePass$^+$ is significantly faster than MultiPass while its result set is close to the exact solution, and (c) in contrast to the other algorithms, ESX is scalable, i.e., it can compute approximate k-SPwLO queries for large road networks and large values of k.

In the future, we plan to extend the definition of alternative routing by considering additional constraints and criteria besides the overlap between paths and their length. We also plan to perform a qualitative study to identify which criteria users value more when deciding upon which route to follow. Finally, we plan to investigate the computation of multiple dissimilar paths on different types of networks such as social networks and web graphs.

8. REFERENCES

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Alternative Routes in Road Networks. *Journal of Experimental Algorithmics*, 18:1–17, 2013.

[2] V. Akgun, E. Erkut, and R. Batta. On finding dissimilar paths. *European Journal of Operational Research*, 121(2):232–246, 2000.

[3] R. Bader, J. Dees, R. Geisberger, and P. Sanders. Alternative Route Graphs in Road Networks. In *Proc. of the 1st Int. ICST Conference on Practice and Theory of Algorithms in Computer Systems (TAPAS)*, pages 21–32, 2011.

[4] A. Bernstein and D. Karger. A nearly optimal oracle for avoiding failed vertices and edges. In *Proc. of the 41st ACM Symposium on Theory of Computing*, pages 101–110, 2009.

[5] Cambridge Vehicle Information Technology Ltd. Choice Routing, 2005.

[6] T. Chondrogiannis, P. Bouros, J. Gamper, and U. Leser. Alternative routing: k-shortest paths with limited overlap. In *Proc. of the 23rd ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems*, pages 68:1–68:4, 2015.

[7] T. Chondrogiannis and J. Gamper. ParDiSP: A partition-based framework for distance and shortest path queries on road networks. In *Proc. of the 17th IEEE Int. Conf. on Mobile Data Management*, pages 242–251, 2016.

[8] D. Delling and W. Dorothea. Pareto Paths with SHARC. In *Proc. of the 8th Symposium on Exterimental Algorithm (SEA)*, pages 125–136, 2009.

[9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[10] B. Fox. K-th shortest paths and applications to the probabilistic networks. *ORSA/TIMS Joint National Mtg.*, 23:B263, 1975.

[11] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies : Faster and simpler hierarchical routing in road networks. In *Proc. of the 7th Int. Workshop on Experimental Algorithms (WEA)*, pages 319–333, 2008.

[12] Y.-J. Jeong, T. J. Kim, C.-H. Park, and D.-K. Kim. A Dissimilar Alternative Paths-search Algorithm for Navigation Services: A Heuristic Approach. *KSCE Journal of Civil Engineering*, 14(1):41–49, 2009.

[13] H.-P. Kriegel, M. Renz, and M. Schubert. Route skyline queries: A multi-preference path planning approach. In *Proc. of the 26th IEEE ICDE*, pages 261–272, 2010.

[14] Y. Lim and H. Kim. A Shortest Path Algorith for Real Road Network based on Path Overlap. *Journal of the Eastern Asia Society for Transportation Studies*, 6:1426 – 1438, 2005.

[15] E. Q. Martins and M. M. Pascoal. A new implementation of yen's ranking loopless paths algorithm. *4OR: A Quarterly Journal of Operations Research*, 1(2):121–133, 2003.

[16] K. Mouratidis, Y. Lin, and M. l. Yiu. Preference queries in large multi-cost transportation networks. In *2010 IEEE 26th ICDE*, pages 533–544, 2010.

[17] A. Paraskevopoulos and C. Zaroliagis. Improved Alternative Route Planning. In *Proc. of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS)*, pages 108–122, 2013.

[18] D. Sacharidis and P. Bouros. Routing directions: keeping it fast and simple. In *Proc. of the 21st ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems*, pages 164–173, 2013.

[19] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *Proc. of the 2008 ACM SIGMOD Conf.*, pages 43–54, 2008.

[20] J. Sankaranarayanan and H. Samet. Distance oracles for spatial networks. In *Proc. of the 25th IEEE ICDE*, pages 652–663, 2009.

[21] C. Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46(4):1–31, 2014.

[22] L. Valiant. The Complexity of Enumeration and Reliability Problems. *Siam Journal of Computing*, 8(3):410–421, 1979.

[23] K. Xie, K. Deng, S. Shang, X. Zhou, and K. Zheng. Finding alternative shortest paths in spatial networks. *ACM Transactions on Database Systems*, 37(4):29:1–29:31, 2012.

[24] J. Y. Yen. Finding the K Shortest Loopless Paths in a Network. *Management Science*, 17(11):712–716, 1971.

[25] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest Path and Distance Queries on Road Networks: Towards Bridging Theory and Practice. In *Proc. of the 2013 ACM SIGMOD Conf.*, pages 857–868, 2013.

Efficient Parallel Spatial Skyline Evaluation Using MapReduce

Wenlu Wang[†], Ji Zhang[†], Min-Te Sun[‡], Wei-Shinn Ku[†]

[†]*Department of Computer Science and Software Engineering, Auburn University, Auburn, AL, USA*
[‡]*Department of Computer Science and Information Engineering, National Central University, Taoyuan, Taiwan, ROC*
{wenluwang, jizhang, weishinn}@auburn.edu, msun@csie.ncu.edu.tw

ABSTRACT

This research presents an advanced MapReduce-based parallel solution to efficiently address spatial skyline queries on large datasets. In particular, given a set of data points and a set of query points, we first generate the convex hull of the query points in the first MapReduce phase. Then, we propose a novel concept called independent regions, for parallelizing the process of spatial skyline evaluation. Spatial skyline candidates in an independent region do not depend on any data point in other independent regions. Thus, we calculate the independent regions based on the input data points and the convex hull of the query points in the second phase. With the independent regions, spatial skylines are evaluated in parallel in the third phase, in which data points are partitioned by their associated independent regions in the map functions, and spatial skyline candidates are calculated by reduce functions. The results of the spatial skyline queries are the union of outputs from the reduce functions. Due to high cost of the spatial dominance test, which requires comparing the distance from data points to all convex points, we propose a concept of pruning regions in independent regions. All data points in pruning regions can be discarded without the dominance test. Our experimental results show the efficiency and effectiveness of the proposed parallel spatial skyline solution utilizing MapReduce on large-scale real-world and synthetic datasets.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Application—*spatial databases*

Keywords

Spatial Skyline Query, MapReduce, Parallel Computation

1. INTRODUCTION

Since the skyline operator was introduced into database research [4], a number of efficient algorithms have been proposed for the skyline evaluation. Bitmap [25], Index [25], NN (Nearest Neighbor) [16] and BBS (Branch-and-Bound Skyline) [19] rely on indices constructed before query processing; while BNL (Block Nested Loop) [4], D&C (Divide and Conquer) [4], SFS (Sort Filter Skyline) [7], and OSPS (Object-based Space Partitioning Skyline) [32] use non-index techniques. Moreover, several studies primarily focus on the skyline query in a variety of problem settings (data residing in a data stream [22] or on mobile devices [14]).

As a novel type of skyline queries, Spatial Skyline Query (SSQ) was proposed to consider the preference of both static and dynamic object attributes in multi-criteria decision-making applications [23]. Unlike skyline queries that only take static object attributes (e.g., rating and price of restaurants) into account, the distance between objects is also calculated as dynamic attributes in the spatial skyline queries. In particular, given a set of data points P and a set of query points Q in a d-dimensional space, spatial skyline queries return a subset of P, in which data points are not spatially dominated by other data points in P. The spatial dominance is defined by using the distance from data points to all query points.

Spatial skyline query is applicable to many applications. Take crisis management applications as an example, we assume that a number of waterborne infectious disease cases were confirmed at different locations, people who live at spatial skyline places with respect to those locations should be alerted and examined first, because there might be higher possibility that these people may have been exposed to contagious water. Travel planning applications are another type of example. People may prefer the spatial skyline hotels with respect to fixed locations of beaches and museums for their vacation. In this case, people would not like to choose a hotel, which is farther from all interesting attractions than other hotels. One more example of spatial skyline query is that people may plan to have dinner with their friends at weekends. They may consider the distance from their homes to the restaurant for the restaurant selection. The restaurants far from all of their homes would not be in the candidate list, because they may want to save time on the road. Thus, giving a list of spatial skyline restaurants is the first step of the restaurant selection.

Two index-based algorithms were proposed to efficiently address the spatial skyline queries [23]. Branch and Bound Spatial Skyline B^2S^2 algorithm searches spatial skyline candidates by visiting an R-tree from top to bottom. Once a spatial skyline is found, B^2S^2 expands the R-tree to access the node which has minimum mindist value, and compares it with all spatial skyline candidates found so far in spatial dominance test. The other method, Voronoi-based Spatial Skyline VS^2 algorithm relies on a Voronoi diagram created over input data points. VS^2 starts with the closest data points to query points, searches in the space by visiting the neighbors of visited data points over the Voronoi diagram. Due to high cost of the spatial dominance test, VS^2 was improved by reducing the number of spatial dominance tests in [24]. In the method, seed skyline

Series ISSN: 2367-2005

10.5441/002/edbt.2017.38

points (a subset of spatial skyline points) can be identified with spatial dominance test.

However, the following problems motivate us to propose a novel parallel solution for spatial skyline evaluation. Firstly, as data grows rapidly, addressing skyline queries on large-scale datasets in a single-node environment becomes impractical. There are increasing number of approaches proposed for processing skyline queries in distributed and/or parallel environments [13]. But none of parallel spatial skyline solutions were observed in literature. Secondly, the distance between moving objects may keep changing. If indices are created at a preprocessing stage, the cost of index maintenance would be unacceptably high. Thirdly, MapReduce framework has been incorporated into parallel solutions for skyline computation [17] [20] [31] and other database applications [30] [26] [18].

Therefore, we propose a novel three-phase MapReduce-based solution, which is able to efficiently address spatial skyline queries on large-scale datasets in this paper. In particular, given a set of data points and a set of query points, we calculate the convex hull of the query points in the first phase. Initially, the query points are evenly partitioned. Each map function accepts a subset of query points, and outputs a local convex hull. Then, the reduce function produces the global convex hull by merging the intermediate results from the map functions. A filtering method can be applied to filter out unqualified data points before the convex hull computation. For example, CG_Hadoop uses skyline algorithms as a filtering method in convex hull evaluation [11]. Moreover, to parallelize the spatial skyline computation, we propose a novel concept, independent regions, in each of which spatial skylines do not depend on any data point outside the independent region. If data points do not fall in any independent region, they can be discarded because they must be spatially dominated by other data points. Thus, our solution produces the independent regions at the second phase. Each map function receives a subset of data points and the convex hull of query points, and generates locally optimized independent regions. Then, the reduce functions output globally optimized independent regions.

With the independent regions in the third phase, map functions associate data points with their independent regions. By using the unique identifiers of independent regions as keys, all data points in an independent region are sent to a reducer after the shuffle phase, and reduce functions find the spatial skylines in independent regions in parallel. Due to high cost of the spatial dominance test, which requires comparing the distance from data points to all convex points, we propose a novel concept, pruning regions, in independent regions. The pruning regions are the areas in which all data points are dominated by other data points. Thus, since a pruning region is defined by a data point, a convex point, and its adjacent convex points, if a data point is in a pruning region, the data point can be discarded without accessing all convex points and calculating the distance from the data point to them. In addition, a data point may fall in more than one independent regions, and there may exist duplicates in spatial skyline candidates. We employ an elimination method in our solution to remove the duplicates with subtle overhead.

In short, the contributions of this study are summarized below:

1. We propose a parallel scheme to efficiently evaluate spatial skyline queries on large datasets using MapReduce.
2. We introduce a concept of independent regions in our solution. The spatial skyline candidates in an independent region do not depend on any data points in other independent regions. With the feature of the independence, spatial skyline queries can be addressed in parallel.
3. We present a concept of pruning regions in independent regions, in order to minimize the cost of the dominance test by avoiding the computation of distance from data points to all convex points.
4. We evaluate the performance of the proposed solution through extensive experiments with large-scale real-world and synthetic datasets.

The rest of this paper is organized as follows. Section 2 surveys related works. The spatial skyline queries and relevant techniques utilized in our solutions are formally defined in Section 3. In Section 4, our advanced solution is presented. The experimental validation of our design is presented in Section 5. We conclude the paper in Section 6.

2. RELATED WORK

In this section, we review previous works related to spatial skyline queries and parallel solutions for general skyline queries.

2.1 Spatial Skyline Queries

As a special case of dynamic skyline queries, Spatial Skyline Queries (SSQ) can be addressed by Block Nested Loop (BNL) [4] and Branch-and-Bound Skyline algorithms (BBS) [19]. In a dynamic skyline query, each object is mapped to another search space by using pre-defined functions. All the objects that are not dominated by other objects in the search space after the mapping are returned from the dynamic skyline queries. BNL algorithm can address the dynamic skyline queries, because it compares every pair of objects in the input dataset, and eliminates the ones that are dominated by any other objects. BNL does not need indices and is efficient over small datasets. But it suffers from I/O access when the input datasets become large. If the size of skyline candidates exceeds the size of available memory space, all the candidates have to be written to a temporary data stream, and read back when they are needed in the next iteration of object comparison. BBS relies on an R-tree to evaluate the general skyline queries; it calculates the $mindist$ of intermediate entries in the R-tree, and searches the space by expanding the entry with the smallest $mindist$. However, BBS does not consider the relation between the input query points and data points.

Motivated by the inefficiency of BNL and BBS, a Branch-and-Bound Spatial Skyline (B^2S^2) algorithm and a Voronoi-based Spatial Skyline (VS^2) algorithm were proposed for spatial skyline evaluation [23]. In addition to considering the properties of the convex hull generated by input query points, B^2S^2 searches the space by visiting an R-tree from top to bottom. Once the first spatial skyline is found, B^2S^2 expands the R-tree with the node which has the minimum $mindist$ value, and checks the dominance between the visited node and all spatial skyline candidates found so far. The process continues until all intermediate nodes potentially containing spatial skylines have been visited. On the other hand, VS^2 builds a Voronoi diagram over input data points. The input data points are organized by their Hilbert values in pages in order to preserve their locality. After completion of convex hull calculation, VS^2 starts with the closest data points to the query points, and searches the space by visiting the neighbors of visited data points over the Voronoi diagram. For every visited data point, VS^2 compares it with all spatial skylines found so far for spatial dominance test. The process continues until all Voronoi cells (or data points) that potentially contain spatial skylines have been visited. Inspired by high cost of the spatial dominance test, VS^2 was improved by reducing the number of spatial dominance tests [24]. In addition to applying sorting techniques, the method is able to identify seed skyline points (a subset of spatial skyline points) without dominance test. Given a set of query points Q and a set of data points P, let $V(p_i)$ be the Voronoi cell of data point $p_i \in P$, the seed skyline points are the points p_i that $V(p_i)$ intersect with the boundary of the convex hull of Q or is inside the convex hull. However, none of the aforementioned methods can address the spatial skyline query in parallel. B^2S^2 requires a pre-structured R-tree and VS^2 needs to build a Voronoi diagram over input data points. Extending their methods to a distributed and/or parallel environment is non-trivial.

2.2 Parallel Skyline Solution

Due to high cost of skyline evaluation, a number of advanced solutions have been proposed to evaluate the general skyline queries in a distributed and/or parallel environment. Balke *et al.* developed a parallel skyline solution over distributed environments [3]. Their method first vertically partitions input datasets in such a manner that each partition keeps object attributes in one dimension. Then, the skyline objects are calculated in parallel, and reported to a central point for a final dominance check. Wu *et al.* designed a parallel skyline method that leverages content-based data partitioning [28]. Their method can avoid unnecessary data access and can progressively produce skylines by using recursive region partitioning and dynamic region encoding mechanisms. Moreover, the incremental scalability is also provided in such a manner that workload can be automatically balanced by distributing objects to new nodes. In addition to random data partitioning methods that can generate similar data distribution in each partition [8] and grid-based data partitioning methods that consider object proximity [2] [21], Vlachou *et al.* proposed an angle-based data partitioning method that partitions objects by their angular coordinates [27]. The average pruning power of objects within a partition can be increased and the number of skyline objects in local skyline calculation can be minimized by applying the angle-based partitioning method. Köhler *et al.* designed a hyperplane-based data partitioning method in order to minimize the local skylines in a partition and achieve efficient local skyline merging [15]. Moreover, a variety of MapReduce-based parallel solutions have been proposed for skyline queries and other database applications. Han *et al.* proposed an advanced skyline algorithm that utilizes Sorted Positional Index Lists (SSPL) to reduce I/O cost [12]. Zhang *et al.* implemented BNL, SFS, and Bitmap algorithms using MapReduce framework [29]. Chen *et al.* applied an angular data partition in their MapReduce-based solution for skyline query evaluation [6]. Eldawy *et al.* developed CG_Hadoop, a suite of MapReduce algorithms, to solve fundamental computational geometry problems, which include convex hull computation [11]. Mullesgaard *et al.* investigated the general skyline queries by using the MapReduce framework. Their method uses bit strings to represent the dominance relation of attributes, and generates independent partition groups for calculating local skyline objects in parallel [17]. Zhang et al. proposed an efficient parallel skyline solution using MapReduce, in which a Partial-presort Grid-based Partition Skyline (PGPS) algorithm was developed to significantly improve the merging skyline computation on large datasets [31]. More importantly, PGPS can be easily incorporated in the shuffle phase of the MapReduce framework with minor overhead. However, our proposed solution targets on spatial skyline queries, which are different from the general skyline queries. None of the partition schemes or computation algorithms above could address the spatial skyline problem. Therefore, we propose a novel partition method and a parallel algorithm which includes independent regions to parallelize the spatial skyline computation and pruning regions to reduce the cost of spatial dominance test.

Table 1 Symbolic notations.

Symbol	Meaning
P, Q	a set of data points and a set of query points
p, q	a data point and a query point
$p.x_i$	the value of data point p in the i^{th} dimension
$\mathbb{R}^d$	a d-dimensional space
h	a hyper-place
S	a half-space
F	a facet of a convex hull
$A_q^{\triangle}$	a set of adjacent convex points of q
$p \prec^Q p'$	p spatially dominates p' with respect to Q
$SSKY(P,Q)$	spatial skylines of P with respect to Q
$CH(Q)$	the convex hull of Q
$DR(p,Q)$	the dominator region of p with respect to Q
$PR(p,q)$	the pruning region generated by p and q
$IR(p,q)$	the independent region generated by p and q
IRP	Independent Regions Pivot, the independent regions are generated by IRP
$lssky$	a set of local spatial skyline candidates
$chsky$	a set of spatial skyline candidates in a convex hull

3. PRELIMINARIES

3.1 Problem Statement

Given a dataset P in a d-dimensional space $\mathbb{R}^d$, an object $p \in P$ can be represented as $p = \{x_1, x_2, ..., x_d\}$ where $p.x_i$ is the value of the object on the i^{th} dimension. $D(.,.)$ denotes a distance metric that obeys the triangle inequality in $\mathbb{R}^d$. The spatial dominatnce relationship and the spatial skyline operator are defined as follows [23]. All notations used in this paper are summarized in Table 1.

Definition (*Spatial Domination*) Given a set of query points Q, and two data points p and p' in $\mathbb{R}^d$, p spatially dominates p' with respect to Q, denoted by $p \prec^Q p'$, if $\forall\, q \in Q,\ D(p,q) \le D(p',q)$ and $\exists\, q' \in Q,\ D(p,q') < D(p',q')$.

Definition (*Spatial Skyline*) Spatial skylines of a set of data points P with respect to a set of query points Q in $\mathbb{R}^d$, denoted by $SSKY(P, Q)$, are a set of data points in P, which are not spatially dominated by any other data point in P with respect to Q.

$$SSKY(P,Q) = \{p\ \in P \mid \nexists\, p' \in P, p \neq p', p' \prec^Q p\} \qquad (1)$$

PROPERTY 1. *If any data point $p \in P$ is a spatial skyline point with respect to a subset of query points $Q' \subset Q$, then p is also a spatial skyline point with respect to Q [23].*

PROPERTY 2. *The set of spatial skyline points of data points P does not depend on any non-convex query points $q \in Q$, $q \notin CH(Q)$, where $CH(Q)$ indicates the convex hull of Q [23]. In other words,*

$$SSKY(P,Q) = SSKY(P, CH(Q)) \qquad (2)$$

Definition (*Dominator Region*) Given a data point $p \in P$, a set of query points Q, and hyper-spheres that center at q_i with radius $D(p, q_i)$, $q_i \in Q$, any data point inside the intersection of the hyper-spheres spatially dominates p with respect to Q. The intersection area that potentially contains data points spatially dominating p with respect to Q is referred to as the dominator region of p, denoted by $DR(p,Q)$.

Dominator Region enables our solution to efficiently eliminate data points by reducing the search space of data points. For example, Figure 1 displays dominator region of a data point p and a set of query points Q. Q has three query points q_1, q_2, and q_3,

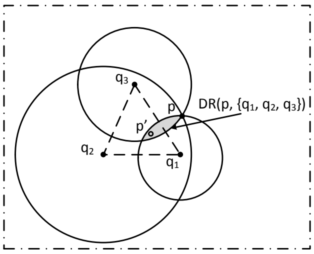


Figure 1: An example of $DR(p, \{q_1, q_2, q_3\})$ in a *2*-dimensional space.

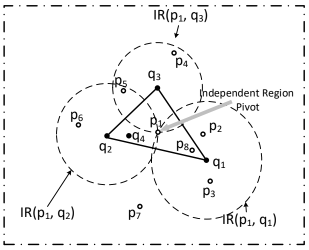


Figure 2: An example of Independent Regions in a *2*-dimensional space.

which represent a convex hull in a *2*-dimensional space. Three circles centered at $q_i \in Q$ with radius $D(q_i, p)$ are created in order to highlight the dominance areas of p with respect to the query points. Any data point p' in the intersection of the three circles spatially dominates p with respect to Q.

3.2 Convex Hull and Spatial Skyline Queries

Given a set of query points Q in a *d*-dimensional space $\mathbb{R}^d$, the convex hull of Q, denoted by $CH(Q)$, is the smallest convex polytope that contains all query points in Q. Theoretically, a convex hull can be represented as either a set of convex points or the intersection of a set of half-spaces. Each half-space contains all the query points in Q. Moreover, a convex hull can also be abstracted by a set of facets and their adjacency relationships. Each facet can be defined by a number of convex points. For example, a facet (line) can be determined by two adjacent convex points in a *2*-dimensional space. The facets become planes that can be represented by a convex point and its two adjacent convex points in a *3*-dimensional space. Because the facets of $CH(Q)$ separate the query points in Q from any point outside the convex hull, connecting a data point v outside $CH(Q)$ with any data point in $CH(Q)$ must intersect with at least one facet of the convex hull. Thus, the facet is referred to as a visible facet from v.

The properties of convex hull provide opportunities to optimize the process of spatial skyline evaluation by reducing the search space of both data points and query points. Given a set of data points P and the convex hull of a set of query points Q, all data points inside $CH(Q)$ are spatial skylines of P with respect to Q [23]. Given two data points, if they are in the convex hull, the bisector hyper-plane of these two points partitions the space into two half-spaces, and there must exist convex points in either half-space. Thus, neither of the two data points can spatially dominate the other, and both of them are spatial skylines. If one point p_1 is in the convex hull and the other p_2 is not, then, the bisector line of p_1 and p_2 partitions the space into two half-spaces, and there must exist a convex point in the same half-space with p_1. If the convex point does not exist, the convex hull cannot contain p_1, which contradicts with our assumption. Thus, p_1 is not spatially dominated by p_2. These two cases are summarized in Property 3.

PROPERTY 3. *Given a set of data points P and a set of query points Q, if any point $p \in P$ is inside the convex hull of Q, then p is a spatial skyline of P with respect to Q ($p \in SSKY(P, Q)$).*

3.3 MapReduce Overview

MapReduce was proposed as a generic programming model for data-intensive applications in distributed environments [10]. The framework provides two simple primitives, map and $reduce$ functions, and allows developers to mainly focus on their functionality. The task scheduling, load balancing, and other issues are encapsulated in the MapReduce framework, which significantly reduces the difficulty of the development of parallel applications. Driven by the MapReduce framework, map functions receive data in key/value pairs from input streams and output intermediate results in another type of key/value pairs. Then, $reduce$ functions retrieve the intermediate results and write final results to an output stream. In the shuffle phase, the intermediate results are automatically grouped and sorted by the MapReduce framework, The two primitives can be represented as: map(K_1, V_1) $\rightarrow$ list(K_2, V_2) and reduce(K_2, list(V_2)) $\rightarrow$ list(K_3, V_3).

4. DESIGN

In this section, we propose our advanced parallel spatial skyline solution using MapReduce. First of all, we briefly present the framework of the solution. Then, our spatial skyline algorithm is illustrated in detail in Section 4.2. The concepts of independent regions and pruning regions are introduced to optimize the process of spatial skyline evaluation. Finally, we discuss three critical implementation issues in our solution.

4.1 Framework Overview

Our solution consists of three MapReduce phases, which receive a set of data points P and a set of query points Q as inputs, and output spatial skyline points of P with respect to Q. As illustrated in Figure 3, we calculate the convex hull of Q in the first MapReduce phase. Q is initially partitioned into subsets of equal size, and each map function finds the local convex hull of query points in a subset. Then, a reduce function generates the global convex hull of Q by merging the local convex hulls. Convex hull algorithm like Graham scan could be employed in each map and reduce function [5]. Due to high complexity of convex hull computation, a filtering method can be used to filter out unqualified points with lower cost. For example, Eldawy *et al.* observed that the convex points must be at least one of four types of skyline points of Q (max-max, min-max, max-min, and min-min) in a *2*-dimensional space, and applied skyline algorithms as a filtering step in their CG_Hadoop system [11].

An intuitive spatial skyline method requires to examine the spatial dominance between every pair of data points. Sharifzadeh and Shahabi utilized the R-tree and Voronoi diagram as indices in their B^2S^2 and VS^2 algorithms [23]. Son *et al.* enhanced VS^2 by reducing the number of dominance tests [24]. However, extending these methods to a parallel solution is non-trivial. Efficiently maintaining indices over data in a distributed and/or parallel environment requires expertise and extensive experience. To address this issue, we propose a novel concept, independent regions, in each of which spatial skyline points do not depend on any data points outside the independent region. With the independence, the input data points can be partitioned by their independent regions, and spatial skyline points can be calculated in parallel. Therefore, after the completion of convex hull computation, we calculate the independent regions based on the convex hull and the input data points P in the second phase. Each map function takes a subset of P and the convex hull of Q as inputs, and outputs a locally optimal **Independent Region Pivot** (See Figure 2, the independent regions are determined by the independent region pivot and convex points). Then a reduce function produces a globally optimal independent region pivot by merging the intermediate results. More details of independent region pivot selection will be discussed in Section 4.3.1. In the third phase, P is initially partitioned, and each map function finds the independent regions of data points in a split. The output of the map functions can be represented as $< IR.id, p >$, where $IR.id$ denotes the unique identifier of the independent region associated with a data point p. There are three possible cases: (1) data points

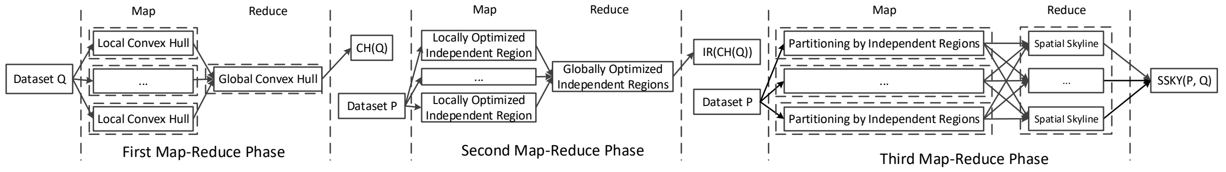


Figure 3: An overview of the parallel spatial skyline processing using MapReduce.

are eliminated if they are outside all independent regions; (2) data points are marked and output as spatial skylines by mappers and reducers if they are inside the convex hull of Q. These data points are needed in reduce functions, because they may spatially dominate data points in category 3; (3) data points are produced with their associated independent regions if they fall in at least one independent region. These data points will be processed by reducers to find spatial skylines in the independent regions. If a data point is inside two or more independent regions, the map function will produce a pair of $< IR.id,\, p >$ for every associated independent region. After the shuffle phase, data points in P are grouped by independent regions, and sent to reduce functions for spatial skyline calculation in parallel. Finally, the global spatial skyline points are the union of the output of reduce functions. A data point could be associated with two or more independent regions, which may introduce duplicates in the results. We design an elimination method to remove duplicates in our solution. The method will be presented in Section 4.3.3.

Figure 2 shows an example of spatial skyline query over three query points and eight data points (Q={q_1, q_2, q_3, q_4}, P={p_1, ..., p_8}). First of all, the convex hull of query points ($CH(Q)$) is generated in the first MapReduce phase. Then, the globally optimal independent region pivot is found by using P and $CH(Q)$ in the second MapReduce phase. Each mapper takes a split of P and $CH(Q)$ (a constant global variable), and selects a local optimal independent region pivot, and a reducer outputs the globally optimal pivot. In the third MapReduce phase, each mapper receives a split of P, and the pivot and $CH(Q)$ (as two constant global variables), and produces object points with their associated independent regions. In the example, there are three independent regions ({$IR(p_1, q_1)$, $IR(p_1, q_2)$, $IR(p_1, q_3)$}). All the independent regions can be calculated from the pivot and $CH(Q)$ in mappers. Moreover, object points are associated with the independent regions where they locate in. If we use ir_1, ir_2, and ir_3 to denote $IR(p_1, q_1)$, $IR(p_1, q_2)$, and $IR(p_1, q_3)$, then p_1 is associated with ir_1, p_5 is associated with ir_2 and etc. After the shuffle phase, $< ir_1, p_1 >$, $< ir_1, p_2 >$, $< ir_1, p_3 >$, and $< ir_1, p_8 >$ are grouped and sent to the first reducer, $< ir_2, p_1 >$, $< ir_2, p_5 >$, and $< ir_2, p_6 >$ are passed to the second reducer, and $< ir_3, p_1 >$ $< ir_3, p_4 >$ $< ir_3, p_5 >$ are processed in the third reducer. In this case, p_1 is a special object point, which is in all three independent regions. As we will discuss our elimination method in Section 4.3.3, p_1 will be only output by the first reducer. Thus, the first reducer outputs p_1, p_2 and p_8 as spatial skylines and discards p_3 because it is dominated by p_8. The second reducer produces p_5, p_6. The third reducer does not output any object because p_4 is eliminated in the spatial dominance test and p_5 has been produced in the second reducer. Finally, the result of the spatial skyline query is the union of the results of reducers, which are {p_1, p_2, p_5, p_6, p_8}.

4.2 Spatial Skyline Calculation

In the second and third MapReduce phases, we generate independent regions based on the convex hull of Q and a set of data points P for spatial skyline computation in parallel. In this subsection, we first provide a formal definition of an independent region. Due to high cost of the spatial dominance test that requires comparing the distance from data points to all convex points, we introduce pruning regions in independent regions. A pruning region can be defined by a data point inside $CH(Q)$, a convex point, and its adjacent convex points. If a data point is in a pruning region, the point can be discarded without the dominance test.

Definition (*Independent Region*) Given a data point p and a set of query points Q in a d-dimensional space, we define an *Independent Region* of p and q_i, $q_i \in Q$ as a sphere centered at q_i with radius $D(p, q_i)$. An **Independent Region Group** (IRG) of p with respect to Q is the union of the independent regions, as shown in Figure 2.

$$\begin{aligned} IRG(p, Q) &= \bigcup_{q_i \in Q} IR(p, q_i), \text{ where} \\ IR(p, q_i) &= \{l \mid D(l, q_i) \le D(p, q_i)\} \end{aligned} \tag{3}$$

We define data point p as the **Independent Region Pivot** of $IRG(p, Q)$ as shown in Figure 2.

With the definition of the independent region, we provide the independence of spatial skylines as follows.

THEOREM 4.1. *Given a data point* p *and its independent regions* $\{IR(p, q_j) \mid q_j \in CH(Q)\}$, *where* $CH(Q)$ *is the convex hull of query points* Q, $\forall\ q_j \in CH(Q)$, *any data point* $p' \in IR(p, q_j)$ *is not dominated by any data point* $p'' \notin IR(p, q_j)$.

PROOF. The proof is by contradiction. Assume that $\exists\ p' \in IR(p, q_j)$, $p'' \notin IR(p, q_j)$, $p'' \prec^Q p'$. By the definition of spatial skyline, p'' is spatially closer to any query point q_i ($q_i \in Q$) than p. Since $q_j \in CH(Q)$, so $q_j \in Q$ as well. But according to the definition of independent regions, $D(p'', q_j) \ge D(p', q_j)$ since p'' is outside of $IR(p, q_j)$, which leads to a contradiction. Thus, this concludes the proof. □

The independent regions are determined by the independent region pivot and the convex hull of Q. The convex hull is uniquely determined by input query points Q; however, theoretically, the pivot can be arbitrarily selected. Since the independent regions specify the search region that contains spatial skyline candidates, an intuitive strategy of the data point selection is to select the data point that minimizes the total volume of its independent regions.

Figure 2 displays an example that utilizes independent regions in the spatial skyline evaluation in $\mathbb{R}^2$. The datasets P and Q consist of 8 data points and 4 query points, respectively. q_1, q_2, and q_3 are the convex points of the convex hull of Q. The three dashed circles indicate three independent regions generated by p_1 and the convex points. In this example, P is partitioned into three subsets, which are $P_1 = \{p_1, p_2, p_3, p_8\}$, $P_2 = \{p_1, p_4, p_5\}$, and $P_3 = \{p_1, p_5, p_6\}$. p_1 and p_8 are spatial skylines, because they are in the convex hull [23]. p_7 is outside all independent regions and can be discarded by mappers in the third phase. p_5 is in $IR(p_1, q_2)$ and $IR(p_1, q_3)$, thus p_5 is associated with both independent regions. Then, the spatial skylines in independent regions are calculated independently. Figure 4 shows an example of the pruning region in $IR(p_1, q_1)$ (the formal definition of pruning regions will be presented in Section 4.2.1). p_8 is a data point that is closer to q_1 than p_1. Thus, we create a pruning region $PR(p_8, q_1)$ (highlighted in gray) in $IR(p_1, q_1)$ to filter out data points dominated by p_8. In the example, p_3 falls in $PR(p_8, q_1)$, and can be discarded without being compared with p_2. Thus, p_2 is the only data point requiring spatial dominance test, comparing its distance to all convex points with the one of p_8. Our spatial skyline algorithm will be presented in Section 4.2.2.

4.2.1 Pruning Regions in Independent Regions

In the third MapReduce phase, a reduce function calculates spatial skylines of a set of data points in an independent region. In particular, the data points are comparing their distances to all convex points of $CH(Q)$ with all other data points (spatial skylines do not depend on non-convex points [23]), and the ones are discarded if they are spatially dominated in the same independent region. The data point comparison would be expensive when the number of convex points of $CH(Q)$ becomes large. Thus, to minimize the cost of the dominance test, we propose a pruning method that is able to efficiently filter out unqualified data points without accessing all convex points of $CH(Q)$. This method defines a pruning region in each independent region; if data points fall in the pruning region, they can be discarded because there must exist a data point dominating these data points. We will first illustrate the pruning regions in a 2-dimensional space, and then provide a formal definition and proof of the pruning regions in high-dimensional spaces.

Figures 6 and 7 show an example of a pruning region in $\mathbb{R}^2$. Given a query point q, two data points p and v, let L_{qx} be a line connecting q to any point $x \in \mathbb{R}^2$, and L_{vq} be the line of q and v. We build a 2-dimensional Cartesian coordinate system, in which q is the origin and L_{qx} is x axis. L_{qx} and L_{vq} partition $\mathbb{R}^2$ into two half-spaces, denoted by S^-_{qx} and S^+_{qx}, and S^-_{vq} and S^+_{vq}, respectively.

THEOREM 4.2. *If p and v satisfy*

$$\begin{aligned} &(1)\ v \in S^+_{qx} \text{ and } p \in S^-_{qx} \\ &(2)\ v.x \le p.x \\ &(3)\ D(v, q) > D(p, q) \end{aligned} \tag{4}$$

then p spatially dominates v with respect to any point q^*, $q^* \in S^-_{qx} \bigcap S^+_{vq}$.

PROOF. As a case of $p.x \ge 0$ shown in Figure 6, the three conditions indicate that (1) L_{qx} partitions v and p into two half-spaces, $v.y > 0$, $p.y < 0$; (2) if we create a line L_{pc} perpendicular to L_{qx}, then v is at the left side of L_{pc}; (3) v is outside of the circle centered at q with radius $D(p, q)$. The circle intersects with L_{pc} at p and b.

Given any data point q^* or q^{**} in $S^-_{qx} \bigcap S^+_{vq}$ (highlighted in gray), L_{vq^*} must intersect with either an arc from a to b (arc_{ab}) or a line from b to c (L_{bc}). If L_{vq^*} intersects with arc_{ab} at e, then we can get $D(p, q) = D(e, q)$ and $p.x > e.x$. Given a query point $q_x = \{q^*.x, 0\}$ on L_{qx}, then

$$\begin{aligned} D(e, q_x) &= \sqrt{(e.x - q^*.x)^2 + (e.y)^2} \\ &= \sqrt{D(e, q)^2 - 2 \cdot (e.x) \cdot (q^*.x) + (q^*.x)^2} \\ &> \sqrt{D(p, q)^2 - 2 \cdot (p.x) \cdot (q^*.x) + (q^*.x)^2} \\ &= \sqrt{(p.x - q^*.x)^2 + (p.y)^2} \\ &= D(p, q_x) \end{aligned} \tag{5}$$

thus, $D(p, q_x) < D(v, q_x)$. If q_x is moved to q^* ($q^*.y < 0$), then $D(p, q^*) < D(v, q^*)$ is also held. On the other hand, if $L_{vq^{**}}$ intersects with L_{bc} at e', then $D(p, q^{**}) < D(e', q^{**})$, because L_{qx} is the bisector line of p and b, and both p and q^{**} are in S^-_{qx}. Thus, $D(p, q^{**}) < D(e', q^{**}) \le D(v, q^{**})$. We can get the similar result in the case of $p.x < 0$. Therefore, p spatially dominates v with respect to any query point in $S^-_{qx} \bigcap S^+_{vq}$. □

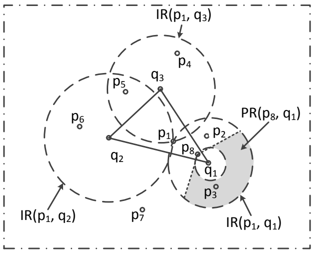


Figure 4: An example of Pruning Regions in $\mathbb{R}^2$.

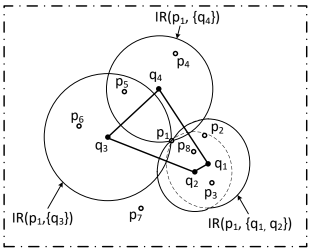


Figure 5: An example of merged Independent Regions in $\mathbb{R}^2$.

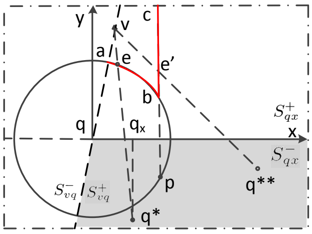


Figure 6: A pruning region using p and a visible facet L_{qx}.

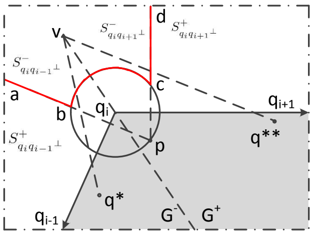


Figure 7: An example of $PR(p, q_i)$.

In a 2-dimensional space, a convex hull is a convex polygon, in which each convex point has two adjacent convex points. Figure 7 shows three convex points q_{i-1}, q_i, and q_{i+1} of a convex hull $CH(Q)$. q_{i-1} and q_{i+1} are adjacent to q_i. Line segments $L_{q_iq_{i-1}}$ and $L_{q_iq_{i+1}}$ are two visible facets from a data point v outside $CH(Q)$ [9].

THEOREM 4.3. *In a 2-dimensional space, given a query point $q_i \in CH(Q)$ and a data point v outside $CH(Q)$, let $A^\triangle_{q_i}$ be a set of adjacent convex points of q_i, and p be an invisible data point from v. Each of the lines from p perpendicular to $L_{q_iq_j}$ ($q_j \in A^\triangle_{q_i}$) partitions the space into two closed half-spaces. Let $S^-_{q_iq_j\perp}$ be the half space containing q_i. Then, any data point v outside $CH(Q)$ satisfying*

$$\begin{aligned} &(1)\ v \in S^-_{q_iq_j\perp},\ q_j \in A^\triangle_{q_i} \\ &(2)\ D(v, q_i) > D(p, q_i) \end{aligned} \tag{6}$$

is spatially dominated by p with respect to Q.

PROOF. In Figure 7, L_{ab} and L_{cd} are two lines from p perpendicular to $L_{q_iq_{i-1}}$ and $L_{q_iq_{i+1}}$, respectively. L_{ab}, L_{cd}, and arc_{bc} separate v from the convex hull $CH(Q)$. L_{vq_i} partitions $CH(Q)$ into two closed half regions, G^- and G^+; all convex points are in either G^- or G^+. If a convex point q^* is in G^-, we can easily get $D(p, q^*) < D(v, q^*)$ by using Theorem 4.2. The similar result can be obtained in the case that any convex point q^{**} is in G^+. Thus, v is spatially dominated by p with respect to Q. □

After an illustration of pruning regions in $\mathbb{R}^2$, we extend the concept of pruning regions to high-dimensional spaces. In a d-dimensional space, a convex hull of query points Q can be represented by a set of half-spaces $\mathcal{H}$, where $CH(Q) = \bigcap_{h^+ \in \mathcal{H}} h^+$. The bisector hyper-plane of each half space contains a *(d-1)*- dimensional facet of the convex hull, which can be determined by a convex point and a subset of its adjacent convex points. The formal definition of the pruning regions is provided as follows.

Definition (*Pruning Regions*) In a d-dimensional space, given a convex hull of query points $CH(Q)$, a data point v outside $CH(Q)$, a visible convex point q, and an invisible data point p from v, let $A_q^{\triangle}$ be a set of adjacent convex points of q in facets, $h_{qq_j}^{\perp}$ be the *(d-1)*-dimensional hyper-plane that contains p and is perpendicular to L_{qq_j}, $q_j \in A_q^{\triangle}$. $h_{qq_j}^{\perp}$ partitions the space into two closed half-spaces; the one containing q is denoted by $S_{h_{qq_j}^{\perp}}^{-}$. Then any data point v outside $CH(Q)$ satisfying

$$\begin{aligned}&(1)\ v \in S_{h_{qq_j}^{\perp}}^{-},\ q_j \in A_q^{\triangle}\\ &(2)\ D(v,q) > D(p,q)\end{aligned} \tag{7}$$

is spatially dominated by p with respect to Q. The region containing all possible v is called a Pruning Region of p and q, denoted by $PR(p,q)$.

PROOF. The proof is by induction. The pruning region in a 2-dimensional space has been proven in Theorem 4.3. We assume that the pruning region is held in an *(i-1)*-dimensional space ($i \geq 3$), then, in an i-dimensional space, since v is outside $CH(Q)$, the line connecting v with any convex point q^* must intersect with a visible closed *(i-1)*-dimensional facet F of $CH(Q)$. The hyper-sphere centered at q with radius $D(p,q)$ and $h_{qq_j}^{\perp}$ ($q_j \in A_q^{\triangle}$) separate v from $CH(Q)$. If a ray from v to q^* intersects with the hyper-sphere at e earlier than any $h_{qq_j}^{\perp}$ ($q_j \in A_q^{\triangle}$), then given any convex point $q_k \in A_q^{\triangle}$, we can build an i-dimensional Cartesian coordinate system, in which F is a hyper-plane (x_i = 0), $v.x_i > 0$, $p.x_i < 0$, and L_{qq_k} ($L_{qq_k} \in F$) is x axis. Any query point on L_{qq_k} can be represented by $\{x_1, 0, ..., 0\}$. Since $D(e,q) = D(p,q)$ and $e.x_1 < p.x_1$, given any query point q_x on L_{qq_k}, we can get

$$\begin{aligned}D(e,q_x) &= \sqrt{(e.x_1 - q_x.x_1)^2 + D(e, L_{qq_k})^2}\\ &= \sqrt{D(e,q)^2 - 2\cdot(e.x_1)\cdot(q_x.x_1) + (q_x.x_1)^2}\\ &> \sqrt{D(p,q)^2 - 2\cdot(p.x_1)\cdot(q_x.x_1) + (q_x.x_1)^2}\\ &= \sqrt{(p.x_1 - q_x.x_1)^2 + D(p, L_{qq_k})^2}\\ &= D(p,q_x)\end{aligned} \tag{8}$$

where $D(e, L_{qq_k})$ denotes the distance from e to the line L_{qq_k}. Thus, we can get that, given any query point q' satisfying D(p, q') $\leq$ D(e, q'), if q' is moved to q'' on any of the directions from q to its adjacent convex points in F, D(p, q'') < D(e, q'') is also held. So, p is closer to any query point in F than e. Since, $v.x_i > 0$, $p.x_i < 0$, and $q^*.x_i < 0$, so $D(p,q^*) < D(v,q^*)$. On the other hand, if a ray from v to q^* first intersects with $h_{qq_k}^{\perp}$ at e', let $q^{\circledast}$ be the center of the intersection of $h_{qq_k}^{\perp}$ and the hyper-sphere centered at q with radius D(p, q), $h_{qq_k}^{\perp}$ is an *(i-1)*-dimensional hyper-plane, in which $D(e', q^{\circledast}) > D(p, q^{\circledast})$, and $e' \in S_{h_{qq_t}}^{-}$, $q_t \neq q_k$, q_t, $q_k \in A_q^{\triangle}$, which satisfies the conditions in an *(i-1)*-dimensional space. Thus, $D(p, q^{\circledast}) < D(e', q^{\circledast})$; then $D(p,q^*) < D(e',q^*) \leq D(v,q^*)$. Therefore, this concludes the proof. □

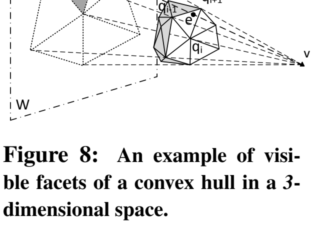


Figure 8: An example of visible facets of a convex hull in a *3*-dimensional space.

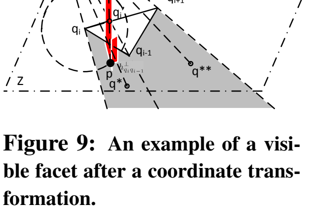


Figure 9: An example of a visible facet after a coordinate transformation.

Figure 8 shows an example of the convex hull of query points Q in a 3-dimensional space. v is a data point outside the convex hull. The line L_{vw_e} intersects with a visible facet F at e. F can be determined by three convex points q_{i-1}, q_i, and q_{i+1}. After a coordinate transformation, F is transformed to be on plane Z ($z = 0$), $v.z > 0$, and $p.z < 0$, as displayed in Figure 9. $L_{q_iq_{i+1}}$ is the x axis in plane Z. The area invisible from v, including p, is highlighted in gray. Two hyper-planes $h_{q_iq_{i-1}}^{\perp}$ and $h_{q_iq_{i+1}}^{\perp}$ perpendicular to $L_{q_iq_{i-1}}$ and $L_{q_iq_{i+1}}$ are highlighted in red. q_{i+1} and q_{i-1} are two elements of $A_{q_i}^{\triangle}$. If a ray from v to q^* first intersects with the sphere centered at q_i with radius $D(p, q_i)$ at e, then according to Equation 8, we can get that given any point q' on $L_{q_iq_{i+1}}$, $q'.x \geq q_i.x$, $D(p,q') \leq D(e,q')$, and moving the point on the direction from q_i to q_{i+1} with distance $\triangle d$ ($\triangle d > 0$) makes the point closer to p than e. The similar result can be obtained on the direction from q_i to q_{i-1}. Thus, any query point in the facet F is closer to p than e. Moreover, $e.z > 0$, $q.z < 0$, and $q^*.z < 0$, we can get that $D(p,q^*) < D(e,q^*) < D(v,q^*)$. On the other hand, if a ray from v to q^{**} first intersects with $h_{q_iq_{i+1}}^{\perp}$ at e', then $e'.x = p.x$, and $D(e', q_i) > D(p, q_i)$. Let q_i' be the intersection of $L_{q_iq_{i+1}}$ and $h_{q_iq_{i+1}}^{\perp}$, $D(e', q_i') > D(p, q_i')$. $h_{q_iq_{i-1}}^{\perp}$ intersects with $h_{q_iq_{i+1}}^{\perp}$ at a line, which contains p and partitions $h_{q_iq_{i+1}}^{\perp}$ into two closed half-spaces. q_i' and e' are in the same half space. By Theorem 4.3, $D(e', q_i') > D(p, q_i')$. Therefore, $D(v, q^{**}) \geq D(e, q^{**}) > D(p, q^{**})$, and v is spatially dominated by p with respect to Q.

4.2.2 *Spatial Skyline Algorithm*

With the concept of pruning regions, we present our spatial skyline algorithm used in reduce functions of the third phase. The input data points are grouped by their independent regions through the shuffle phase, and unqualified data points outside independent regions have been discarded in map functions. The fundamental idea of our method is to first eliminate data points by using pruning regions. If they are not in any pruning region, they are needed to compare with all other potential spatial skyline candidates for spatial dominance test.

The details of our method are described in Algorithm 1. The algorithm receives all data points in an independent region $IR(p, q_i)$, denoted by P_i, and the convex hull of query points Q. We use $chsky$ and $lssky$ to keep local spatial skylines inside and outside $CH(Q)$, respectively. PR abstracts pruning regions of the spatial skyline candidates. The union of $chsky$ and $lssky$ are output as spatial skylines in the independent region, which is a subset of the global spatial skylines of the query.

In particular, the algorithm first finds all the data points in $CH(Q)$. These data points are kept in $chsky$, and used to build pruning regions PR (from lines 4 to 11). $lssky$ temporarily maintains all data points outside $CH(Q)$. Then, each data point in $lssky$ is visited for the dominance test (from lines 12 to 20). If a data point falls in any pruning region, the data point will be removed from $lssky$. If the data point is outside the pruning regions, it needs to

Algorithm 1 Spatial Skyline Algorithm

```
Input: $P_i$, $CH(Q)$
Output: $lssky \cup chsky$
 1: $lssky = \emptyset$;
 2: $chsky = \emptyset$;
 3: $PR = \emptyset$;
 4: for $\forall p \in P_i$ do
 5:     if $p$ is inside $CH(Q)$ then
 6:         $chsky = chsky \cup \{p\}$;
 7:         $PR = PR \cup PR(p, q_i)$;
 8:     else
 9:         $lssky = lssky \cup \{p\}$;
10:     end if
11: end for
12: for $\forall p \in lssky$ do
13:     if $p$ is in $PR$ then
14:         $lssky = lssky - \{p\}$;
15:         Continue;
16:     end if
17:     if $\exists\, p' \in (chsky \cup lssky),\ p' \neq p,\ p' \prec^Q p$ then
18:         $lssky = lssky - \{p\}$;
19:     end if
20: end for
21: return $lssky \cup chsky$;
```

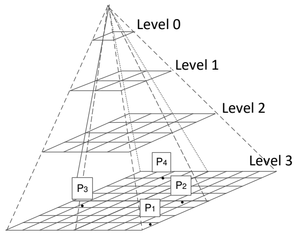


Figure 10: An example of $Grid(lssky \cup chsky)$**.**

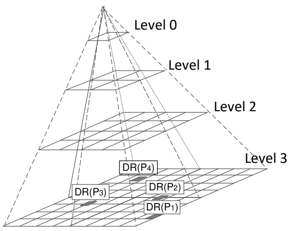


Figure 11: An example of $Grid(DR(lssky \cup chsky))$**.**

compare with all other data points in $chsky$ and $lssky$, and will be eliminated if it is dominated.

To minimize the cost of the dominance test in line 17, we use two multi-level grids, $Grid(lssky \cup chsky)$ and $Grid(DR(lssky \cup chsky))$, to maintain spatial skyline candidates and their dominator regions (defined in Section 3.1). The two grids are always synchronized; once there is a data point inserted into or removed from $Grid(lssky \cup chsky)$, $Grid(DR(lssky \cup chsky))$ is updated accordingly. Figures 10 and 11 display an example of the two grids. The cells at the bottom level keep the references of spatial skyline candidates and their dominator regions; parent cells maintain the proximity information of their child cells. In the dominance test of a new data point p, we first check if p is dominated by other data points. We calculate the dominator region of p, and visit $Grid(lssky \cup chsky)$ from top to bottom to see if there is a data point falling in the dominator region. The iteration can stop at any intermediate level when either of the two conditions is satisfied: (1) all cells intersecting with the dominator region do not contain any data point (p is not dominated by spatial skyline candidates in $lssky \cup chsky$); (2) a cell inside the dominator region contains a data point (p is dominated by the data point). If p is not dominated, then we visit $Grid(DR(lssky \cup chsky))$ in a similar manner to see if p dominates any data point in $lssky \cup chsky$. If p falls in the dominator region of p', then p' and its dominator region will be removed from both $Grid(lssky \cup chsky)$ and $Grid(DR(lssky \cup chsky))$.

4.3 Implementation Issues

In this subsection, we discuss three implementation issues in our solutions.

4.3.1 *Independent Region Pivot Selection*

In the second MapReduce phase, the search space is partitioned into a number of independent regions. The spatial skylines are calculated in parallel by reducers in the third MapReduce phase. The execution time of a parallel program is determined by the slowest process, and the spatial skyline algorithm takes longer on larger inputs. Thus, distributing the data points to reducers in a balanced manner is critical to our approach.

If the data points are uniformly distributed in the search space, the number of data points in an independent region is proportional to the volume of the independent region, which depends on the distance between the independent region pivot and the convex point. Theoretically, the point with equal distance to all convex points is the optimal independent region pivot, which could split data points in equal size. But the optimal pivot may not exist in irregular convex hull. Moreover, the point that minimizes the total volume of independent regions would be an alternative optimal pivot. However, the cost of finding the point is expensive. Thus, we turn to an approximation method in our implementation. After the convex hull is calculated, we choose the center of the Minimum Bounding Rectangle (MBR) of the convex hull as the independent region pivot. The experimental results of varying independent region pivots can be found in Section 5.6.

4.3.2 *Independent Region Merging*

In the third phase of our solution, a reducer processes data points in an independent region. The number of reducers needed in the spatial skyline calculation depends on the number of independent regions or the number of convex points in the convex hull of query points Q. Since the size of the convex hull would be large, the task maintenance and communication overhead in MapReduce framework would be unacceptably high.

Thus, there are two merging strategies that can be applied to our proposed solution if the number of independent regions is much greater than the number of available computing resources. In the strategies, we assume that objects are uniformly distributed in the search space. The smaller the total volume of independent regions is, the less the number of objects are processed in spatial skyline computation.

Shortest distance merging. In this method, we merge the closest pair of two neighboring independent regions. The distance of two independent regions is evaluated by the distance between the centers of the independent regions. We assume that there is higher possibility that two independent regions overlap with each other if they are close. Merging two overlapped independent regions may reduce the cost of spatial skyline computation for the following two reasons: (1) the objects in the overlapping region are fed to one reducer instead of two, which minimizes the total number of objects in the spatial dominance test; (2) the pruning regions of the independent regions are also merged; more objects could be eliminated without the dominance test. Take Figure 5 for example, q_1 and q_2 are the closest pair of convex points in the figure. $IR(p_1, q_1)$ and $IR(p_1, q_2)$ are merged, and the new independent region is denoted by $IR(p_1, \{q_1, q_2\})$. So, p_3 and p_8 are only processed by the reducer, which receives $IR(p_1, \{q_1, q_2\})$. The pruning region of $IR(p_1, \{q_1, q_2\})$ is PR(p_1, q_1) ∪ PR(p_1, q_2).

In our implementation, we iterate the convex hull in counter clockwise order, and calculate the distance between every pair of two consecutive independent regions. Let n be the number of computing resources available to the spatial skyline evaluation and m

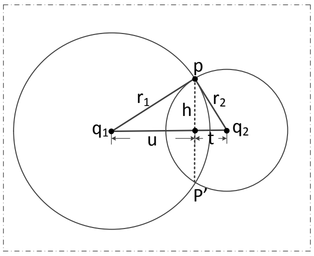


Figure 12: Volume of overlapping region of two independent regions.

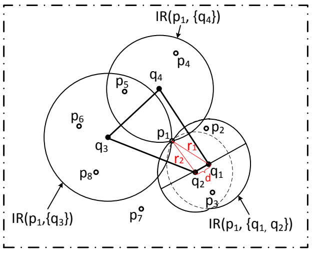


Figure 13: An example of independent regions in a *2*-dimensional space.

be the number of convex points, we will merge the top $m - n$ ($m \geq n$) closest pairs of the independent regions (the number of pairs of independent regions is equal to the number of convex points).

Threshold-based merging. An alternative strategy merges independent regions by considering the volume of the overlapping region of two independent regions. In this method, we visit the independent regions in counter clockwise order. Given two consecutive independent regions, we calculate the ratio of volume of the overlapping region of the two independent regions to the volume of the smaller independent region. If the ratio is higher than a specific threshold, the two independent regions will be merged. Another difference from the first method is that two or more independent regions may be merged if they are close to each other. The ratio can be defined as follows.

$$ratio(q_1, q_2) = \frac{Vol^d(IR(p_1, q_1)) \cap Vol^d(IR(p_1, q_2))}{Vol^d(IR(p_1, q_2))} \tag{9}$$

where $IR(p_1, q_1)$ and $IR(p_1, q_2)$ are two consecutive independent regions, $Vol^d(IR(p_1, q_2))$ denotes the volume of $IR(p_1, q_2)$ in a d-dimensional space, and $Vol^d(IR(p_1, q_1)) \geq Vol^d(IR(p_1, q_2))$.

Moreover, the volume of the overlapping region of two independent regions (two spheres) can be calculated as follows (See Figures 12 and 13).

$$\begin{aligned} &Vol^d(IR(p_1, q_1) \cap IR(p_1, q_2)) = \\ &\qquad \int_{u_0}^{r_1} Vol^{d-1}(h)du + \int_{t_0}^{r_2} Vol^{d-1}(h)dt \end{aligned} \tag{10}$$

where $Vol^{d-1}(h)$ denotes the volume of the sphere with radius h in a *(d-1)*-dimensional space, $h = (r_1^2 - u^2)^{1/2} = (r_2^2 - t^2)^{1/2}$. u_0 and t_0 are the lower bounds of the integrals, where $u_0 = \frac{r_1^2 - r_2^2 + D(q_1, q_2)^2}{2D(q_1, q_2)}$ and $t_0 = \frac{r_2^2 - r_1^2 + D(q_1, q_2)^2}{2D(q_1, q_2)}$. $D(q_1, q_2)$ denotes the distance between q_1 and q_2.

Figure 12 shows an example of the independent region merging in a 2-dimensional space. Line $l_{pp'}$ decomposes the overlapping region into two sub-regions. The length of $l_{pp'}$ is denoted by $V^1(h)$ = $2h$. If we move $l_{pp'}$ towards q_1 and q_2, respectively, then, the volume of the two sub-regions is the sum of integral of $V^{d-1}(h)$ in the overlapping area. In a d-dimensional space, $l_{pp'}$ becomes a sphere in a *(d-1)*-dimensional hyper-plane, and h is the radius of the sphere.

In a 2-dimensional space, $ratio(q_1, q_2)$ can be calculated as follows,

$$\begin{aligned} ratio(q_1, q_2) &= \frac{Vol^2(IR(p, q_1)) \cap Vol^2(IR(p, q_2))}{Vol^2(IR(p, q_1))} \\ &= \frac{\int_{u_0}^{r_1}(h)du + \int_{t_0}^{r_2}(h)dt}{Vol^2(IR(p, q_1))} \\ &= \frac{r^2 cos^{-1}(\frac{d^2 + r_1^2 - r_2^2}{2dr_2}) + r^1 cos^{-1}(\frac{d^2 + r_2^2 - r_1^2}{2dr_1})}{\pi {r_1}^2} \end{aligned} \tag{11}$$

4.3.3 Duplicate Elimination

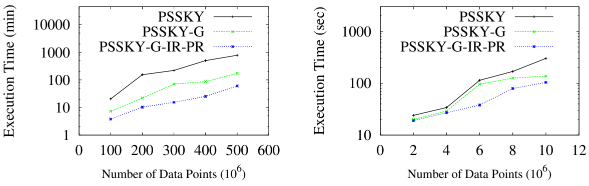
The third issue is that our solution may produce duplicates since a data point may locate in two or more independent regions. If the data point is a spatial skyline, it will be written to the results of the query by multiple reducers. To eliminate the duplicates, we associate a unique independent region identifier to each data point, which indicates that the data point will be output as a spatial skyline by the reducer which processes data points in the independent region. Reducers processing data points in other independent regions will not output the data point even if it is a spatial skyline. Take Figure 4 as an example, p_5 is a data point in $IR(p_1, q_2)$ and $IR(p_1, q_3)$. If the identifier of $IR(p_1, q_2)$ is associated with p_5, and p_5 is a spatial skyline, p_5 is output only by the reducer processing data points in $IR(p_1, q_2)$.

5. EXPERIMENTAL VALIDATION

In this section, we evaluate the performance of the proposed MapReduce-based solution over synthetic and real-word datasets. Our proposed algorithm is denoted by $PSSKY$-G-IR-PR, which combines the concepts of independent regions, pruning regions, and multi-level grid data structure for efficient query evaluation. Since none of the existing solutions can be easily extended to address spatial skyline queries in parallel, we developed two single-phase MapReduce-based solutions as baselines, $PSSKY$ and $PSSKY$-G. $PSSKY$ applies a random data partitioning method to split data points. Each mapper uses BNL to produce local spatial skylines by comparing every pair of data points, and a reducer merges the local results and outputs the global spatial skylines. $PSSKY$-G works similarly to $PSSKY$ except that $PSSKY$-G utilizes multi-level grid data structure for efficient spatial dominance test. Since all three solutions use the same algorithm in convex hull computation, we will focus primarily on the investigation of the overall performance of solutions and the effect of independent regions and pruning regions on spatial skyline computation in the second and third MapReduce phases. All solutions were implemented in Java on Hadoop 2.6, which is an open source implementation of the MapReduce framework [1].

In the experiments, the real-word datasets were downloaded from Geonames [1]. We retrieved 11 million objects (streams, schools, etc.) in the United States, and used them as data points and query points. The data points in our synthetic datasets are randomly generated under uniform distribution in a 2-dimensional space. Similarly with [23], the query points were also generated in a specified region at the center of the search space. By default, there are 10 convex points in the convex hull of query points. The area covered by the Minimum Bounding Rectangle (MBR) of query points is fixed at 1% of the search space. The experiments were conducted on a 12-node shared-nothing cluster. Each node is equipped with 19 Intel Xeon 2.2 GHz processors and 128 GBytes of memory. All nodes are connected by GigaBit Ethernet network. All results were recorded after the system model reached a steady state.

[1] http://www.geonames.org/



(a) Synthetic datasets. (b) Real-world datasets.

Figure 14: The overall execution time of the three solutions by varying dataset cardinality.

5.1 Scalability with Cardinality

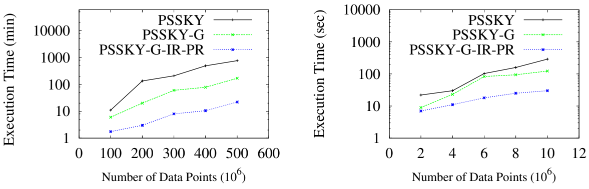
First of all, we evaluate the effect of data cardinality on all three solutions. We vary the cardinality of both synthetic and real-world datasets from 100 to 500 million and 2 to 10 million data points, respectively. As Figure 14 displays, the execution time of all solutions increase when the datasets grow. The growth rate of $PSSKY\text{-}G\text{-}IR\text{-}PR$ over synthetic datasets is lower than $PSSKY$ and $PSSKY\text{-}G$. In addition, on average, $PSSKY\text{-}G\text{-}IR\text{-}PR$ executes around 90% faster than $PSSKY$ and 32% faster than $PSSKY\text{-}G$, respectively. The reason is that $PSSKY\text{-}G\text{-}IR\text{-}PR$ is able to parallelize the spatial skyline evaluation by applying the concept of independent regions and efficiently filter out unqualified data points in pruning regions. Moreover, a performance improvement was observed when comparing $PSSKY\text{-}G$ with $PSSKY$, because the multi-level grid data structure is employed to efficiently access the proximity information of data points for the dominance test.

5.2 Effect of Independent Regions and Pruning Regions on Spatial Skyline Algorithms

To evaluate the effectiveness of independent regions and pruning regions on the query evaluation, we compare the execution time of spatial skyline computation in $PSSKY\text{-}G\text{-}IR\text{-}PR$ (the execution time of reducers in the third MapReduce phase) with the ones in $PSSKY$ and $PSSKY\text{-}G$. The cardinality of synthetic and real-world datasets varies from 100 to 500 million and 2 to 10 million data points.As Figure 15 shows, the execution time of all solutions increase when the datasets grow. The execution time of $PSSKY$ increases rapidly due to high complexity of spatial skyline computation. The growth rate of $PSSKY\text{-}G\text{-}IR\text{-}PR$ is the lowest, because all data points can be processed in parallel and a significant portion of data points can be discarded without dominance test. Moreover, the reducer that merges spatial skylines becomes a bottleneck in $PSSKY$ and $PSSKY\text{-}G$, which consumes 50% to 90% of the total execution time over large synthetic and real-world datasets.

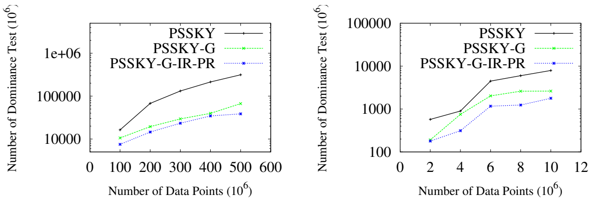
5.3 Effect of Number of Nodes

We evaluate the speedup of proposed solutions by scaling up the size of our cluster. The real and uniform datasets are fixed at 10 million and 100 million objects. The cardinality of cluster nodes varies from 2 to 12.



(a) Synthetic datasets. (b) Real-world datasets.

Figure 15: The execution time of spatial skyline algorithms by varying dataset cardinality.



(a) Synthetic datasets. (b) Real-world datasets.

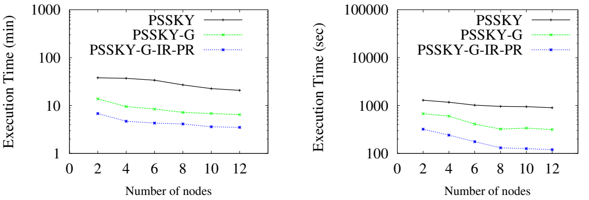
Figure 16: The number of dominance test by varying dataset cardinality.

In Figure 17, the execution time of all solutions drops as the size of the cluster increases. As expected, $PSSKY$ always consumes more execution time than $PSSKY\text{-}G$ and $PSSKY\text{-}G\text{-}IR\text{-}PR$ while scaling up the cluster. On average, $PSSKY\text{-}G\text{-}IR\text{-}PR$ enjoys the highest dropping rate. Take experiments on real world data for example, $PSSKY\text{-}G\text{-}IR\text{-}PR$ drops 34.35% when scaling up to 8 nodes, PSSKY-G only drops 27%; the dropping rate of $PSSKY$ is constantly lower than 20% in all experiments. The reason is that more map or reduce tasks can be executed in parallel with more computing resources. However, even all three methods will take advantage of mapper parallelism, only reducers of $PSSKY\text{-}G\text{-}IR\text{-}PR$ run in parallel, because the global region is partitioned into independent regions; the skyline results in each independent region do not depend on the ones in other independent regions.

For both synthetic and real data, $PSSKY\text{-}G\text{-}IR\text{-}PR$ performs approximately 50% better than $PSSKY\text{-}G$ and 80% better than $PSSKY$ in terms of execution time.

5.4 Effect of Pruning Regions

We also evaluate the effect of pruning regions by comparing the number of dominance tests among three solutions. The cardinality of synthetic and real-world datasets varies from 100 to 500 million and 2 to 10 million data points. Figure 16 displays the number of dominance test in the three solutions over the datasets. As expected, $PSSKY$ always suffers from more dominance tests than $PSSKY\text{-}G$ and $PSSKY\text{-}G\text{-}IR\text{-}PR$. Using multi-level grid data structure can reduce the cost of the dominance test, because, instead of access all data points, $PSSKY\text{-}G$ only needs to visit data points in the cells that intersect with dominator regions of other data points. Moreover, the effect of pruning regions can be observed by comparing the results of $PSSKY\text{-}G$ and $PSSKY\text{-}G\text{-}IR\text{-}PR$. Although data points locating at two or more independent regions may introduce subtle overhead in data point comparison, $PSSKY\text{-}G\text{-}IR\text{-}PR$ can save more time in the dominance test by utilizing the concept of pruning regions. According to our experiments, there are a small number of duplicate data points generated



(a) Synthetic datasets. (b) Real-world datasets.

Figure 17: The overall execution time of the three solutions by varying nodes cardinality.

Table 2: Effectiveness of pruning regions by varying dataset cardinality.

Synthetic DataSets	Number of Data Points (million)				
	100	200	300	400	500
	27%	27%	27%	27%	27%

Real-world DataSets	Number of Data Points (million)				
	2	4	6	8	10
	10%	10%	9%	9%	8%

by $PSSKY\text{-}G\text{-}IR\text{-}PR$. Combining the experiments shown in Figure 15, $PSSKY\text{-}G\text{-}IR\text{-}PR$ only takes a few minutes longer if there are additional 100 million dominance test are performed by $PSSKY\text{-}G\text{-}IR\text{-}PR$. Note that these dominance tests may be conducted in parallel in $PSSKY\text{-}G\text{-}IR\text{-}PR$.

Table 2 shows the power of pruning regions in terms of data point reduction rate by varying cardinality of datasets. The reduction rate is defined by the average percentage of data points eliminated by pruning regions in independent regions. We find that around 27% of data points in synthetic datasets fall in pruning regions, and can be discarded by $PSSKY\text{-}G\text{-}IR\text{-}PR$ while there are around 9% of real-world data points that can be pruned without the dominance test. Moreover, the object elimination rate is slightly changed for large real-world datasets. The reason is that the number of data points varies in the experiments, but the number of query points and its convex hull are fixed. Theoretically, if data points are uniformed distributed, the effectiveness of the Pruning Region only depends on the volume of Pruning Regions. Increasing the density of data points over a large data point datasets does not help much to generate larger pruning regions. Thus, the reduction rate over synthetic datasets remains unchanged, and there is slight change in the reduction rate over real-world datasets due to the non-uniform distribution of data points.

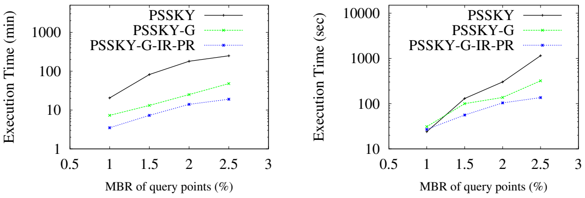
Table 3 shows the power of pruning regions in terms of data point reduction rate by varying the distribution of data points. We replace 5%, 10%, 15%, and 20% of uniform data points with anti-correlated data points. For example, the experiments performed over datasets with 20% anti-correlated and 80% uniform data points are displayed in the first row of the results in Table 3. We find that the reduction rate remains the same over datasets under the same distribution of data points. Moreover, when 20% of anti-correlated data points are generated in datasets, only 2% difference is observed in the experiments, which tells us that the ratio of the volume of independent regions and pruning regions to that of the search space is small. In other words, if 20% data points are moved to the central area of the search space, there are only 2% of data points moved outside the pruning regions.

5.5 Effect of Query Points

We investigate the effect of the area covered by the convex hull of query points on the solutions in this subsection. We fix the size of data points at 100 million. The ratio of the area covered by the MBR of query points to the search space ranges from 1% to 2.5%. The number of Convex Hull query points selected for real-world datasets are 10, 14, 17, and 23, and that for synthetic datasets are 10, 12, 14, and 16, respectively. Figure 18 displays the overall execution time of solutions over synthetic and real-world datasets. The ratios of the MBR of the convex hull of query points are indicated by x axis. Intuitively, a larger convex hull may help to reduce the cost of the dominance test because more data points would locate in the convex hull, and can be output as spatial skylines without dominance test. However, our experimental results show that the entire process of query evaluation takes longer. The reason is that there are more data points in the search region, and the number of data points requiring dominance test becomes larger. Take Figure 2 for example, a convex hull is represented by q_1, q_2, and q_3. p_1 is used to generate three independent regions. If the convex hull grows larger, the area covered by the three independent regions will become larger accordingly. Thus, more data will be located in the independent regions, and be processed by reducers in the third MapReduce phase.

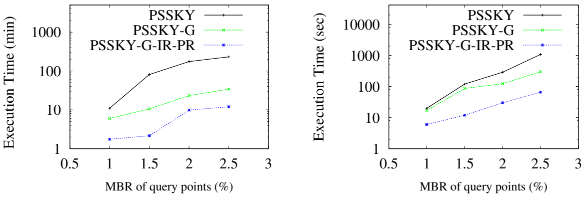
Table 3: Effectiveness of pruning regions by varying dataset distribution.

DataSets	Number of Data Points (million)				
	100	200	300	400	500
20% anti-correlated	24%	24%	24%	24%	24%
15% anti-correlated	24.7%	24.7%	24.7%	24.7%	24.7%
10% anti-correlated	25.3%	25.3%	25.3%	25.3%	25.3%
5% anti-correlated	26%	26%	26%	26%	26%



(a) Synthetic datasets. (b) Real-world datasets.

Figure 18: The overall execution time of the three solutions by varying the MBR of the convex hull of query points.



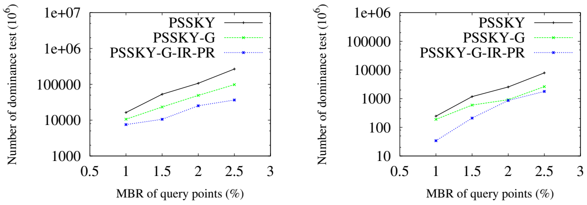
(a) Synthetic datasets. (b) Real-world datasets.

Figure 19: The execution time of spatial skyline algorithms by varying the MBR of the convex hull of query points.

The evidence is also displayed in Figure 19 and 20. Figure 19 shows the execution time of spatial skyline computation and the number of dominance tests grow rapidly when the MBRs of the convex hull of query points cover larger areas. A similar results are observed in terms of the number of dominance test in Figure 20.

5.6 Effect of Independent Region Pivot Selection

We investigate the effect of independent region pivot selection on the query evaluation by varying the pivot on real world datasets. We



(a) Synthetic datasets. (b) Real-world datasets.

Figure 20: The number of dominance test in the three solutions by varying the MBR covered by query points.

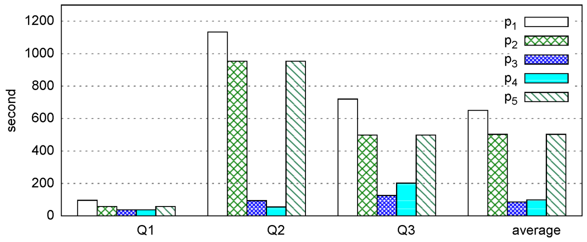


Figure 21: The execution time of $PSSKY$-G-IR-PR by varying independent region pivots.

randomly choose three query dataset in such a way that the convex hull of the query points is in random shape. Figure 20 shows the execution time of the query when the pivot locates at the query points with minimum and maximum y-coordinate values (p_5 and p_1), and the center of $MBR(CH(Q))$ (p_3). Two additional pivots are selected at the midpoint of the line of p_1 and p_3, and the line of p_3 and p_5. The two pivots are denoted by p_2 and p_4, respectively. In general, p_3 is closer to the optimal pivot than other four pivots, and $PSSKY$-G-IR-PR runs faster when p_3 is selected as the independent region pivot.

6. CONCLUSION AND FUTURE WORK

In this paper, we propose an advanced parallel spatial skyline solution utilizing MapReduce framework. Given a set of data points and a set of query points, our approach first calculates the convex hull of the query points. Then, we propose a novel concept of independent regions; the input data points are partitioned by their associated independent regions. Unqualified data points outside independent regions can be eliminated without the dominance test. Finally, all spatial skylines in independent regions are calculated in parallel, and the global spatial skylines are the union of local spatial skylines. Moreover, to avoid high cost of data point comparison, we propose a concept of pruning regions, in which objects can be discarded without comparing their distance to all convex hull query points. We demonstrate the efficiency and effectiveness of the proposed solution through extensive experiments on large-scale real-world, and synthetic datasets.

We plan to extend the proposed parallel solution to address spatial skyline queries on road networks. Theoretically, the concepts of independent regions and pruning regions can be applied in the space of road networks. However, more investigation is needed to evaluate the cost of calculating the independent regions and pruning regions. Due to variety of data distribution, it is also interesting to study the pruning power of pruning regions on road networks.

Acknowledgments

This research has been funded in part by the National Science Foundation grants IIS-1618669 (III) and ACI-1642133 (CICI).

References

[1] Apache Hadoop. `http://hadoop.apache.org`.

[2] F. N. Afrati, P. Koutris, D. Suciu, and J. D. Ullman. Parallel skyline queries. In *ICDT*, pages 274–284, 2012.

[3] W.-T. Balke, U. Güntzer, and J. X. Zheng. Efficient Distributed Skylining for Web Information Systems. In *EDBT*, pages 256–273, 2004.

[4] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, pages 421–430, 2001.

[5] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete & Computational Geometry*, 10(1):377–409, 1993.

[6] L. Chen, K. Hwang, and J. Wu. Mapreduce skyline query processing with a new angular partitioning approach. In *IPDPS Workshops*, pages 2262–2270, 2012.

[7] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with Presorting. In *ICDE*, pages 717–719, 2003.

[8] A. Cosgaya-Lozano, A. Rau-Chaplin, and N. Zeh. Parallel Computation of Skyline Queries. In *HPCS*, page 12, 2007.

[9] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

[10] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[11] A. Eldawy, Y. Li, M. F. Mokbel, and R. Janardan. CG_Hadoop: Computational Geometry in MapReduce. In *SIGSPATIAL*, pages 294–303, 2013.

[12] X. Han, J. Li, D. Yang, and J. Wang. Efficient Skyline Computation on Big Data. *IEEE Trans. Knowl. Data Eng.*, 25(11):2521–2535, 2013.

[13] K. Hose and A. Vlachou. A survey of skyline processing in highly distributed environments. *VLDB J.*, 21(3):359–384, 2012.

[14] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline Queries Against Mobile Lightweight Devices in MANETs. In *ICDE*, page 66, 2006.

[15] H. Köhler, J. Yang, and X. Zhou. Efficient parallel skyline processing using hyperplane projections. In *SIGMOD Conference*, pages 85–96, 2011.

[16] D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *VLDB*, pages 275–286, 2002.

[17] K. Mullesgaard, J. L. Pedersen, H. Lu, and Y. Zhou. Efficient Skyline Computation in MapReduce. In *EDBT*, 2014.

[18] A. Okcan and M. Riedewald. Processing theta-joins using MapReduce. In *SIGMOD Conference*, pages 949–960, 2011.

[19] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

[20] Y. Park, J.-K. Min, and K. Shim. Parallel computation of skyline and reverse skyline queries using mapreduce. *PVLDB*, 6(14):2002–2013, 2013.

[21] J. B. Rocha-Junior, A. Vlachou, C. Doulkeridis, and K. Nørvåg. AGiDS: A Grid-Based Strategy for Distributed Skyline Query Processing. In *Globe*, pages 12–23, 2009.

[22] A. D. Sarma, A. Lall, D. Nanongkai, and J. J. Xu. Randomized multi-pass streaming skyline algorithms. *PVLDB*, 2(1):85–96, 2009.

[23] M. Sharifzadeh and C. Shahabi. The Spatial Skyline Queries. In *VLDB*, pages 751–762, 2006.

[24] W. Son, M. Lee, H. Ahn, and S. Hwang. Spatial Skyline Queries: An Efficient Geometric Algorithm. In *SSTD*, pages 247–264, 2009.

[25] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient Progressive Skyline Computation. In *VLDB*, pages 301–310, 2001.

[26] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD Conference*, pages 495–506, 2010.

[27] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *SIGMOD Conference*, pages 227–238, 2008.

[28] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. El Abbadi. Parallelizing Skyline Queries for Scalable Distribution. In *EDBT*, pages 112–130, 2006.

[29] B. Zhang, S. Zhou, and J. Guan. Adapting skyline computation to the mapreduce framework: Algorithms and experiments. In *DASFAA Workshops*, pages 403–414, 2011.

[30] C. Zhang, F. Li, and J. Jestes. Efficient parallel kNN joins for large data in MapReduce. In *EDBT*, pages 38–49, 2012.

[31] J. Zhang, X. Jiang, W.-S. Ku, and X. Qin. Efficient Parallel Skyline Evaluation using MapReduce. *IEEE Trans. Parallel Distrib. Syst.*, 2015.

[32] S. Zhang, N. Mamoulis, and D. W. Cheung. Scalable skyline computation using object-based space partitioning. In *SIGMOD Conference*, pages 483–494, 2009.

Vertex-Centric Graph Processing: Good, Bad, and the Ugly

Arijit Khan
Nanyang Technological University, Singapore
arijit.khan@ntu.edu.sg

ABSTRACT

We study distributed graph algorithms that adopt an iterative vertex-centric framework for graph processing, popularized by Google's Pregel system. Since then, there are several attempts to implement many graph algorithms in a vertex-centric framework, as well as efforts to design optimization techniques for improving the efficiency. However, to the best of our knowledge, there has not been any systematic study to compare these vertex-centric implementations with their sequential counterparts. Our paper addresses this gap in two ways. (1) We analyze the computational complexity of such implementations with the notion of time-processor product, and benchmark several vertex-centric graph algorithms whether they perform more work with respect to their best-known sequential solutions. (2) Employing the concept of balanced practical Pregel algorithms, we study if these implementations suffer from imbalanced workload and large number of iterations. Our findings illustrate that with the exception of Euler tour tree algorithm, all other algorithms either perform asymptotically more work than their best-known sequential approach, or suffer from imbalanced workload/ large number of iterations, or even both. We also emphasize on graph algorithms that are fundamentally difficult to be expressed in vertex-centric frameworks, and conclude by discussing the road ahead for distributed graph processing.

1. INTRODUCTION

In order to achieve low latency and high throughput over massive graph datasets, data centers and cloud operators consider scale-out solutions, in which the graph and its data are partitioned horizontally across cheap commodity servers in the cluster. The distributed programming model for large graphs has been popularized by Google's Pregel framework [4], which was inspired by the Bulk Synchronous Parallel (BSP) model [12]. It hides distribution related details such as data partitioning, communication, underlying system architecture, and fault tolerance behind an abstract API. In Pregel, also known as the *think-like-a-vertex* model, graph algorithms are expressed as a sequence of iterations called supersteps. During a superstep, Pregel executes a user-defined function for each vertex in parallel. The user-defined function specifies the operation at a single vertex v and at a single superstep S. The supersteps are globally synchronous among all vertices, and messages are usually sent along the outgoing edges from each vertex.

With the inception of the Pregel framework, vertex-centric distributed graph processing has become a hot topic in the database community (for a survey, see [13]). Although Pregel provides a high-level distributed programming abstract, it suffers from efficiency issues such as the overhead of global synchronization, large volume of messages, imbalanced workload, and straggler problem due to slower machines. Therefore, more advanced vertex-centric models (and its variants) have been proposed, e.g., asynchronous (GraphLab), asynchronous parallel (GRACE), barrierless asynchronous parallel (Giraph Unchained), data parallel (GraphX, Pregelix), gather-apply-scatter (PowerGraph), timely dataflow (Naiad), and subgraph centric frameworks (NScale, Giraph++). Various algorithmic and system-specific optimization techniques were also designed, e.g., graph partitioning and re-partitioning, combiners and aggregators, vertex scheduling, superstep sharing, message reduction, finishing computations serially, among many others.

While speeding up any algorithm is always significant in its own right, there may be circumstances in which we would not benefit greatly from doing so. McSherry et. al. [5] empirically demonstrated that single-threaded implementations of many graph algorithms using a high-end 2014 laptop are often an order of magnitude faster than the published results for state-of-the-art distributed graph processing systems using multiple commodity machines and hundreds of cores over the same datasets. Surprisingly, with the exception of [14], the complexity of vertex-centric graph algorithms has never been formally analyzed. As one may realize, this is not a trivial problem — there are multiple factors involved in a distributed environment including the number of processors, computation time, network bandwidth, communication volume, and memory usage. To this end, we make the following contributions.

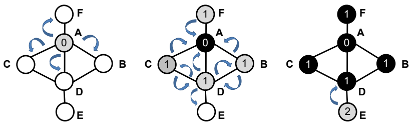
- We formally analyze the computational complexity of vertex-centric implementations with the notion of time-processor product [12], and benchmark several vertex-centric graph algorithms whether they perform asymptotically more work in comparison to their best-known sequential algorithms.
- We use the concept of balanced, practical Pregel algorithms [14] to investigate if these vertex-centric algorithms suffer from imbalanced workload and large number of iterations.

While the notion of balanced, practical Pregel algorithms was introduced by Yan et. al. [14], they only considered the connected component-based algorithms. On the contrary, in this paper we report as many as fifteen different graph algorithms (Table 1), whose vertex-centric algorithms were implemented in the literature. Finally, we also identify graph workloads and algorithms that are difficult to be expressed in the vertex-centric framework, and highlight some important research directions.

10.5441/002/edbt.2017.39

	Graph	Vertex-Centric		Best Sequential		Vertex-Centric	
	Workload	Algorithm	Complexity	Algorithm	Complexity	More Work?	BPPA?
1	Diameter (Unweighted)	[6]	$\mathcal{O}(mn)$	BFS [9]	$\mathcal{O}(mn)$	**No**	No
2	PageRank [1]	[4]	$\mathcal{O}(mK)$	power iteration	$\mathcal{O}(mK)$	**No**	No
3	Connected Component	Hash-Min [4]	$\mathcal{O}(m\delta)$	BFS [3]	$\mathcal{O}(m+n)$	Yes	No
4	Connected Component	S-V [14]	$\mathcal{O}((m+n)\log n)$	BFS [3]	$\mathcal{O}(m+n)$	Yes	No
5	Bi-Connected Component	[14]	$\mathcal{O}((m+n)\log n)$	DFS [3]	$\mathcal{O}(m+n)$	Yes	No
6	Weakly Connected Component	[14]	$\mathcal{O}((m+n)\log n)$	BFS [3]	$\mathcal{O}(m+n)$	Yes	No
7	Strongly Connected Component	[14]	$\mathcal{O}((m+n)\log n)$	DFS [11]	$\mathcal{O}(m+n)$	Yes	No
8	**Euler Tour of Tree**	[14]	$\mathcal{O}(n)$	DFS	$\mathcal{O}(n)$	**No**	**Yes**
9	Pre- & Post-order Tree Traversal	[14]	$\mathcal{O}(n\log n)$	DFS	$\mathcal{O}(n)$	Yes	**Yes**
10	Spanning Tree	[14]	$\mathcal{O}((m+n)\log n)$	BFS	$\mathcal{O}(m+n)$	Yes	No
11	Minimum Cost Spanning Tree [1]	[10]	$\mathcal{O}(\delta m\log n)$	Chazelle's algorithm	$\mathcal{O}(m\alpha(m,n))$	Yes	No
12	Betweenness Centrality (Unweighted)	[8]	$\mathcal{O}(mn)$	Brandes' algorithm	$\mathcal{O}(mn)$	**No**	No
13	Single-Source Shortest Path	[4]	$\mathcal{O}(mn)$	Dijkstra with Fibonacci heap	$\mathcal{O}(m+n\log n)$	Yes	No
14	All-pair Shortest Paths (Unweighted)	[6]	$\mathcal{O}(mn)$	Chan's algoithm	$\mathcal{O}(mn)$	**No**	No
15	Graph Simulation [1]	[1]	$\mathcal{O}(m^2(n_q+m_q))$	Henzinger et. al. [2]	$\mathcal{O}((m+n)(m_q+n_q))$	Yes	No

Table 1: Efficiency analysis for vertex-centric graph algorithms: # nodes = n, # edges = m, diameter = δ



(a) Superstep 0 (b) Superstep 1 (c) Superstep 2

Figure 1: Vertex-centric algorithm for diameter computation in unweighted graphs

2. PRELIMINARIES

2.1 Time-Processor Product

Time-processor product was employed by Valiant [12] as a complexity measure of algorithms on the BSP model, defined by the following parameters. (1) Bandwidth parameter is g, that measures the permeability of the network to continuously send traffic to uniformly-random destinations. The parameter g is defined such that an h-relation will be delivered in time hg. (2) Synchronization periodicity is L, where the components at regular intervals of L time units are synchronized. In a superstep of periodicity L, L local operations and $\lfloor L/g \rfloor$-relation message patterns can be realized. (3) The number of processors is p. Let w_i be the amount of local work performed by processor i in a given superstep. Assume s_i and r_i be the number of messages sent and received, respectively, by processor i. Let $w = \max_{i=1}^{p} w_i$, and $h = \max_{i=1}^{p}(\max(s_i, r_i))$. Then, the time for a superstep is $\max(w, gh, L)$.

If we have multiple processors, we can solve a problem more quickly by dividing it into independent sub-problems and solving them at the same time, one at each processor. Given an input size n, the running time $T(n)$ is the elapsed time from when the first processor begins executing to when the last processor stops executing. A BSP algorithm for a given problem is called efficient if its processor bound $P(n)$ and time bound $T(n)$ are such that time-processor product $P(n)T(n) = \mathcal{O}(S)$, where S is the running time of the best-known sequential algorithm for the problem, provided that L and g are below certain critical values. Therefore, with this metric, we measure whether a vertex-centric algorithm performs more work, compared to the problem's best-known sequential algorithm.

2.2 Balanced, Practical Pregel Algorithms

For an undirected graph, we denote by $d(v)$ the degree of vertex v. On the other hand, let $d_{in}(v)$ and $d_{out}(v)$ denote the in-degree and out-degree, respectively, of vertex v in a directed graph. A Pregel algorithm is called a balanced, practical Pregel algorithm (BPPA) [14] if it satisfies the following. (1) Each vertex v uses $\mathcal{O}(d(v))$ (or, $\mathcal{O}(d_{in}(v)+d_{out}(v))$) storage. (2) The time complexity of the vertex-compute() function for each vertex v is $\mathcal{O}(d(v))$ (or, $\mathcal{O}(d_{in}(v)+d_{out}(v))$). (3) At each superstep, the size of the messages sent/received by each vertex v is $\mathcal{O}(d(v))$ (or, $\mathcal{O}(d_{in}(v)+d_{out}(v))$). (4) The algorithm terminates after $\mathcal{O}(\log n)$ supersteps. Properties 1-3 offer good load balancing and linear cost at each superstep, whereas property 4 impacts the total running time.

3. COMPLEXITY ANALYSIS

We summarize the complexity of fifteen vertex-centric graph algorithms in Table 1. We shall discuss five of them in the following.

3.1 Diameter Computation

We consider a vertex-centric algorithm [6] that computes the exact diameter of an unweighted graph. Let us denote the eccentricity $\epsilon(v)$ of a vertex v as the largest hop-count distance from v to any other vertex in the graph. The diameter δ of the graph is defined as the maximum eccentricity over all its nodes. Instead of finding this largest vertex eccentricity one-by-one, the algorithm works by computing the eccentricity of all vertices simultaneously.

We illustrate in Figure 1 the eccentricity computation method of one vertex. Initially, each vertex adds it's own unique id to the outgoing messages (sent along the outgoing edges) and also to the history set, which resides in the local memory of that vertex. After the initial superstep, the algorithm operates by iterating through the set of received ids, which correspond to the vertices that sent the original messages. The receiving vertex then constructs a set of outgoing messages by adding each element of the incoming set which was not seen yet. The reason for keeping a history of the originating ids that were received earlier is to prevent the re-propagation of a message to the same vertices. The history set also serves to prune the set of total messages by eliminating message paths that would never result in the vertex's eccentricity.

Assuming the graph is connected, each vertex will process a message from each originating vertex exactly once. The algorithm terminates when the largest eccentricity is calculated; and therefore, the diameter of the graph is equal to the number of supersteps (minus 1, for the final, non-processing superstep).

Since each vertex generates a unique message, there are total $\Theta(n)$ messages. Each message is passed $\mathcal{O}(m)$ times, resulting in a message complexity of $\mathcal{O}(mn)$. There are total $\mathcal{O}(\delta)$ supersteps. Each vertex processes n messages; therefore, the overall computation cost is $\mathcal{O}(n^2)$. Assuming bandwidth parameter [2] $g = \mathcal{O}(1)$, the time-processor product = $\mathcal{O}(mn)$, which is equal to the complexity of the best-known sequential algorithm.

However, this vertex-centric algorithm is not BPPA because: (1) The number of messages that each vertex v relays can be asymptotically larger than $\mathcal{O}(d(v))$ at later supersteps. (2) Given that each vertex v must store a history of the messages received, each vertex stores $\mathcal{O}(n)$ vertex IDs, which is larger than $\mathcal{O}(d(v))$. (3) There are total $\mathcal{O}(\delta)$ supersteps, which could be larger than $\mathcal{O}(\log n)$.

[1] K is # iterations for convergence, $\alpha()$ functional inverse of Ackermann's function. n_q and m_q the number of nodes and edges, respectively, in the query graph.

[2] For higher values of g, the time-processor product would be even higher.

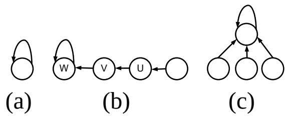
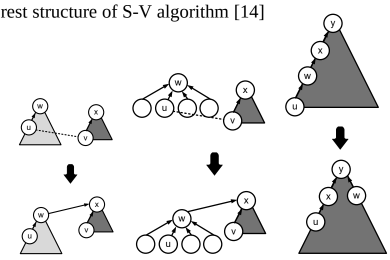


Figure 2: Forest structure of S-V algorithm [14]



(a) Tree-hooking (b) Star-hooking (c) Shortcutting

Figure 3: Tree hooking, star hooking, and shortcutting [14]

3.2 Connected Component

We study two vertex-centric algorithms: Hash-min and S-V [14].

3.2.1 Hash-Min Algorithm

We assume that each vertex in a graph G is assigned a unique ID. The color of a connected component in G is defined as the smallest vertex among all vertices in the component. In Superstep 1, each vertex v initializes $min(v)$ as the smallest vertex in the set $(\{v\} \cup neighbors(v))$, sends $min(v)$ to all v's neighbors, and votes to halt. In each subsequent superstep, a vertex v obtains the smallest vertex from the incoming messages, denoted by u. If $u < v$, v sets $min(v) = u$ and sends $min(v)$ to all its neighbors. Finally, v votes to halt. When all vertices vote to halt and there is no new message in the network, the algorithm terminates.

It takes at most $\mathcal{O}(\delta)$ supersteps for the ID of the smallest vertex to reach all the vertices in a connected component, and in each superstep, each vertex v takes at most $\mathcal{O}(d(v))$ time to compute $min(v)$ and sends/receives $\mathcal{O}(d(v))$ messages each using $\mathcal{O}(1)$ space. Therefore, it is a balanced Pregel algorithm (i.e., satisfies properties 1-3), but not BPPA since the number of supersteps can be larger than $\mathcal{O}(\log n)$, e.g., for a straight-line graph.

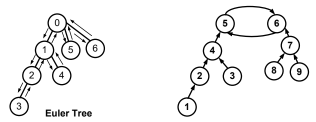
Each superstep consists of $\mathcal{O}(m)$ messages and $\mathcal{O}(m)$ computations. Assuming $g = \mathcal{O}(1)$, the time-processor product is $\mathcal{O}(m\delta)$. This is more than the complexity of the best-known sequential algorithm, which is due to BFS with complexity $\mathcal{O}(m + n)$.

3.2.2 Shiloach-Vishkin (S-V) Algorithm

In the S-V algorithm, each vertex u maintains a pointer $D[u]$. Initially, $D[u] = u$, forming a self-loop as depicted in Figure 2(a). During the algorithm, vertices are arranged by a forest such that all vertices in each tree in the forest belong to the same connected component. The tree definition is relaxed a bit to allow the tree root w to have a self-loop (see Figures 2(b) and 2(c)), i.e., $D[w] = w$; while $D[v]$ of any other vertex V in the tree points to v's parent.

The S-V algorithm proceeds in iterations, and in each iteration, the pointers are updated in three steps (Figure 3): (1) *tree hooking*: for each edge (u, v), if u's parent $w = D[u]$ is a tree root, hook w as a child of v's parent $D[v]$ (i.e., merge the tree rooted at w into v's tree); (2) *star hooking*: for each edge (u, v), if u is in a star (see Figure 2(c) for an example of star), hook the star to v's tree as Step (1) does; (3) *shortcutting*: for each vertex v, move vertex v and its descendants closer to the tree root, by hooking v to the parent of v's parent, i.e., setting $D[v] = D[D[v]]$. The algorithm terminates when every vertex is in a star. We perform tree hooking in Step (1) and star hooking in Step (2) only if $D[v] < D[u]$, which ensures that the pointer values monotonically decrease.

It was proved that the above S-V algorithm computes connected components in $\mathcal{O}(\log n)$ supersteps [14]. However, the algorithm is not a BPPA because a vertex v may become the parent of more than $d(v)$ vertices and hence receives/sends more than $d(v)$ messages in a superstep. On the other hand, the overall number of messages and computations in each superstep are bounded by $\mathcal{O}(n)$ and $\mathcal{O}(m)$, respectively. With $g = \mathcal{O}(1)$, we have the time-processor product = $\mathcal{O}((m + n) \log n)$. As earlier, this is higher than the complexity of the best-known sequential algorithm.



(a) Euler tour (b) Conjoined-tree: vertex 5 is super-vertex

Figure 4: Euler Tour and MCST construction

3.3 Euler Tour Tree Traversal

A Euler tour is a representation of a tree, where each tree edge (u, v) is considered as two directed edges (u, v) and (v, u). As shown in Figure 4(a), a Euler tour of the tree is simply a Eulerian circuit of the directed graph, that is, a trail that visits every edge exactly once, and ends at the same vertex where it starts.

We assume that the neighbors of each vertex v are sorted according to their IDs, which is usually common for an adjacency list representation of a graph. For a vertex v, let $first(v)$ and $last(v)$ be the first and last neighbor of v in that sorted order; and for each neighbor u of v, if $u \neq last(v)$, let $next_v(u)$ be the neighbor of v next to u in the sorted adjacency list. We also define $next_v(last(v)) = first(v)$. As an example, in Figure 4(a), $first(0) = 1$, $last(0) = 6$, $next_0(1) = 5$, and $next_0(6) = 1$.

Yan et. al. [14] designed a 2-superstep vertex-centric algorithm to construct the Euler tour as given below. In Superstep 1, each vertex v sends message $\langle u, next_v(u)\rangle$ to each neighbor u; in Supertep 2, each vertex u receives the message $\langle u, next_v(u)\rangle$ sent from each neighbor v, and stores $next_v(u)$ with v in u's adjacency list. Thus, for every vertex u and each of its neighbor v, the next edge of (u, v) is obtained as $(v, next_v(u))$, which is the Euler tour.

The algorithm requires a constant number of supersteps. In every superstep, each vertex v sends/receives $\mathcal{O}(d(v))$ messages, each using $\mathcal{O}(1)$ space. By implementing $next_v(.)$ as a hash table associated with v, we can obtain $next_v(u)$ in $\mathcal{O}(1)$ expected time given u. Therefore, the algorithm is BPPA. In addition, with $g = \mathcal{O}(1)$, the time-processor product = $\mathcal{O}(n)$. This matches with the time complexity of the best-known sequential algorithm.

3.4 Minimum Cost Spanning Tree

Salihoglu et. al. implemented the parallel (vertex-centric) version of Boruvka's minimum cost spanning tree (MCST) algorithm [10] for a weighted, undirected graph G. The algorithm iterates through the following phases, each time adding a set of edges to the MCST S it constructs, and removing some vertices from G until there is just one vertex, in which case the algorithm halts.

1. Min-Edge-Picking: In parallel, the edge list of each vertex is searched to find the minimum weight edge from that vertex. Ties are broken by selecting the edge with minimum destination ID. Each picked edge (v, u) is added to S. As proved in Boruvka's algorithm, the vertices and their picked edges form disjoint subgraphs $T_1, T_2, \ldots, T_k$, each of which is a *conjoined-tree*, i.e., two trees, the roots of which are joined by a cycle (Figure 4(b)). We refer to the vertex with the smaller ID in the cycle of T_i as the super-vertex of T_i. All other vertices in T_i are called its sub-vertices. The following steps merge all of the sub-vertices of every T_i into the super-vertex of T_i.

2. Super-vertex Finding: First, we find all the super-vertices. Each vertex v sets its pointer to the neighbor v picked in Min-Edge-Picking. Then, it sends a message to v.pointer. If v finds that it received a message from the same vertex to which it sent a message earlier, it is part of the cycle. The vertex with the smaller ID in the cycle is identified as the super-vertex. After this, each vertex finds the super-vertex of the conjoined-tree it belongs to using the *Simple Pointer Jumping* algorithm. The input R to the algorithm is the set of super-vertices, and the input S is the set of sub-vertices.

```
Simple-Pointer-Jumping-Algorithm $(R, S)$
  repeat until every vertex in $S$ points to a vertex in $R$
    for each vertex $v$ that does not point to a vertex in $R$ do
      perform a pointer jump: $v$.pointer → $v$.pointer.pointer
```

3. Edge-Cleaning-and-Relabeling: We shrink each conjoined tree into the super-vertex of the tree. This is performed as follows. In the set of edges of G, each vertex is renamed with the ID of the super-vertex of the conjoined tree to which it belongs. The modified graph may have self-loops and multiple edges. All self-loops are removed. Multiple edges are removed such that only the lightest edge remains between a pair of vertices.

The above operations can be implemented in $\mathcal{O}(\delta)$ supersteps, which is due to the maximum number of iterations required for the simple pointer jumping algorithm. Each superstep has message and computation complexity $\mathcal{O}(m)$. The three above phases are repeated, that is, the graph remaining after the i-th iteration is the input to the $i+1$-th iteration, unless it has just one vertex, in which case the algorithm halts. Furthermore, the number of vertices of the graph at the $i+1$-th iteration is at most half of the number of vertices at the i-th iteration. Hence, the number of iterations is at most $\mathcal{O}(\log n)$. With $g = \mathcal{O}(1)$, the time-processor product is $\mathcal{O}(m\delta \log n)$. This is higher than the complexity of the best-known sequential algorithm for MCST, which is $\mathcal{O}(m\alpha(m,n))$ by Chazelle's algorithm. Here, $\alpha()$ is the functional inverse of Ackermann's function, and it grows extremely slowly, so that for all practical purposes it may be considered a constant no greater than 4. Even if we consider widely-used Prim's algorithm (sequential), it has time complexity $\mathcal{O}(m + n\log n)$ using fibonacci heap and adjacency list. In summary, the vertex-centric algorithm for MCST performs more work than the problem's sequential solutions.

The algorithm is not in BPPA, since (1) the Edge-Cleaning-and-Relabeling step increases the number of neighbors of the super-vertices, and (2) the number of supersteps is $\mathcal{O}(\delta \log n)$.

3.5 Difficult Problems for Vertex-Centric Model

An important question is whether *all* kinds of graph analytics tasks and algorithms can be expressed *efficiently* at vertex level. (1) Vertex-centric model usually operates on the entire graph, which is often not necessary for online ad-hoc queries [15], including shortest path, reachability, and subgraph isomorphism. (2) This model is not well-suited for graph analytics that require a subgraph-centric view around vertices, e.g., local clustering coefficient, triangle and motifs counting. This is due to the communication overhead, network traffic, and the large amount of memory required to construct multi-hop neighborhood in each vertex's local state [7]. (3) Not all distributed algorithms for the same graph problem can be implemented in a vertex-centric framework. As an example, it is difficult to implement the distributed union-find algorithm for the connected component problem using a vertex-centric model [5]. However, this algorithm is useful for graph streams. (4) State-of-the-art research on vertex-centric graph processing mainly focused on a limited number of graph workloads such as PageRank and connected components, and it is largely unknown whether some other widely-used graph computations, e.g., modularity optimization for community detection, betweenness centrality (weighted graphs), influence maximization, link prediction, partitioning, and embedding can be implemented efficiently over vertex-centric systems.

4. DISCUSSION AND CONCLUSION

Our analysis shows that vertex-centric algorithms often suffer from imbalanced workload/ large number of iterations, and perform more work than their best-known sequential algorithms.

Due to such difficulties, alternate proposals exist where the entire graph is loaded on a single machine having larger memory, or on a multi-core machine with shared-memory. Nevertheless, distributed graph processing systems would still be critical due to the two following reasons. First, graph analysis is usually an intermediate step of some larger data analytics pipeline, whose previous and following steps might require distribution over several machines. In such scenarios, distributed graph processing would help to avoid expensive data transfers. Second, distributed-memory systems generally scale well, compared to their shared-memory counterparts.

However, one distributed model might not be suitable for all kinds of graph computations. Many recent distributed systems, e.g., Trinity, NScale, and Apache Flink support multiple paradigms, including vertex-centric, subgraph-centric, dataflow, and shared access. But, perhaps more importantly, we need to identify the appropriate metrics to evaluate these systems. In addition to time-processor product and BPPA that we studied in this work, one can also investigate the speedup and cost/computation. Two other critical metrics are *expressibility* and *usability*, which were mostly ignored due to their qualitative nature. The former identifies the workloads that can be efficiently implemented in a distributed framework, while the later deals with ease in programming, e.g., domain-specific languages, declarative programming, high-level abstraction to hide data partitioning, communication, system architecture, and fault tolerance, as well as availability of debugging and provenance tools. With all these exciting open problems, this research area is likely to get more attention in the near future.

5. REFERENCES

[1] A. Fard, M. U. Nisar, L. Ramaswamy, J. A. Miller, and M. Saltz. A Distributed Vertex-Centric Approach for Pattern Matching in Massive Graphs. In *IEEE International Conference on Big Data*, 2013.

[2] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing Simulations on Finite and Infinite Graphs. In *FOCS*, 1995.

[3] J. Hopcroft and R. Tarjan. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Commun. ACM*, 16(6):372–378, 1973.

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD*, 2010.

[5] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at What Cost? In *HOTOS*, 2015.

[6] C. Pennycuff and T. Weninger. Fast, Exact Graph Diameter Computation with Vertex Programming. In *HPGM*, 2015.

[7] A. Quamar, A. Deshpande, and J. Lin. NScale: Neighborhood-centric Analytics on Large Graphs. In *VLDB*, 2014.

[8] M. Redekopp, Y. Simmhan, and V. K. Prasanna. Optimizations and Analysis of BSP Graph Processing Models on Public Clouds. In *IPDPS*, 2013.

[9] L. Roditty and V. V. Williams. Fast Approximation Algorithms for the Diameter and Radius of Sparse Graphs. In *STOC*, 2013.

[10] S. Salihoglu and J. Widom. Optimizing Graph Algorithms on Pregel-like Systems. In *VLDB*, 2014.

[11] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[12] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, 1990.

[13] D. Yan, Y. Bu, Y. Tian, A. Deshpande, and J. Cheng. Big Graph Analytics Systems. In *SIGMOD*, 2016.

[14] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees. In *VLDB*, 2014.

[15] Q. Zhang, D. Yan, and J. Cheng. Quegel: A General-Purpose System for Querying Big Graphs. In *SIGMOD*, 2016.

Top-*k* Skyline Groups Queries

Haoyang Zhu*, Peidong Zhu*, Xiaoyong Li*†, Qiang Liu*
* College of Computer,
† Academy of Ocean Science and Engineering,
National University of Defense Technology, Changsha, China
{zhuhaoyang, pdzhu, sayingxmu}@nudt.edu.cn, qiangl.ne@hotmail.com

ABSTRACT

The top-k skyline groups query (k-SGQ) returns k skyline groups that dominate the maximum number of points in a given data set. It combines the advantages of skyline groups and top-k queries. The k-SGQ is an important tool for queries that need to analyze not only *individual* points but also *groups* of points, and can be widely used in areas such as decision support applications, market analysis and recommendation system. In this paper, we formally define this new problem and design an efficient algorithm to solve this problem. Extensive experimental results show that our algorithm is effective and efficiency.

1. INTRODUCTION

The skyline query [1] is widely used in multi-criteria optimal decision making applications, which aims at retrieving points that are not dominated by other points in a data set. In this paper, we assume that **larger** values are preferred. Q^i denotes the i^{th} point and Q^i_k denotes the value on the k^{th} dimension of Q^i, then Q^i dominates Q^j, denoted as $Q^i \prec Q^j$, *iff* for each k, $Q^i_k \geq Q^j_k$ and for at least one k, $Q^i_k > Q^j_k$ $(1 \leq k \leq d)$. Fig. 1 shows a skyline example. The data set in Fig. 1 (left) consists of 5 points. Each point has two dimensions. We can see that $Q^4(4,4) \prec Q^3(4,2)$ as an example of dominance relationship between points. As shown in Fig. 1 (right), the skyline contains Q^1, Q^2 and Q^4.

Though skyline computation is particularly useful in multi-criteria decision making applications, it is inadequate to answer queries that need to analyze not only *individual* points but also their combinations [3, 4, 2, 6, 9]. Specifically, in many real-world applications, we need to find groups of points that are not dominated by other groups of equal size.

It is shown in [3, 4, 2, 6, 9] that a skyline group may consist of both skyline points and non-skyline points, all points in the data set have a chance to form a skyline group. Therefore, there are total C^k_n combinations, which are far more than the n candidates in traditional skyline computation. Moreover, the output size of skyline groups is far more than the size of skyline. The experimental results proposed in [6, 9] show that the output size is million scale even when the input is a few thousand points. The large output size is less informative and it may be hard for users to make a good, quick selection.

points	A_1	A_2
Q^1	6	0
Q^2	0	6
Q^3	4	2
Q^4	4	4
Q^5	0	5

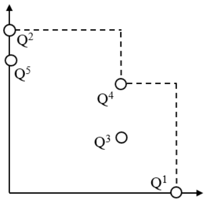


Figure 1: A skyline example

Motivation. The large output size promotes us to design an algorithm to select the best k skyline groups. Such k skyline groups should be most representative. Inspired by the top-k skyline queries, we quantify the concept of "representative" by counting the number of points dominated by the group. Here, we define that a point Q is dominated by a group G *iff* there exists at least one point Q' in G satisfying $Q' \prec Q$. We briefly summarize our contributions as follows:

- We propose a novel problem, top-k skyline groups query, so that the k skyline groups with maximal number of dominated points can be produced to facilitate user queries.
- We propose an efficient algorithm for processing k-SGQ, using several pruning techniques.
- We conduct extensive experiments to validate the effectiveness and efficiency of our proposals.

2. PRELIMINARY

In this section, we introduce the problem definition and related works.

2.1 Problem Definition

First, we introduce the definitions of dominance relationship between groups defined in [3, 4, 2, 6, 9]. We use $\prec_g$ to denote the dominance relationship between groups. Let $G \prec_g G'$ denote G dominates G'. The dominance relationship between groups defined in existing works can be divided into two kinds.

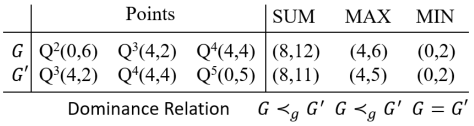
Definition 1. ($\prec_g$) [6] Assuming that $G = \{Q^1, Q^2, ..., Q^l\}$ and $G' = \{Q'^1, Q'^2, ..., Q'^l\}$ are two different groups with l

points. We say that $G \prec_g G'$, *iff* there exist two permutations of the l points for G and G', $G = \{Q^{u1}, Q^{u2}, ..., Q^{ul}\}$ and $G' = \{Q'^{u1}, Q'^{u2}, ..., Q'^{ul}\}$ satisfying that for each i, $Q^{ui} \preceq Q'^{ui}$ and for at least one i, $Q^{ui} \prec Q'^{ui}$ $(1 \leq i \leq l)$.

For instance in the Fig.1, since $Q^2 \prec Q^5$ and $Q^4 \prec Q^3$, thus $\{Q^2, Q^4\} \prec_g \{Q^3, Q^5\}$.

Definition 2. **($\prec_g$)** [3, 4, 2, 9] For an aggregate function f and a group $G = \{Q^1, Q^2, ..., Q^l\}$, then G is represented by a point Q, where $Q_j = f(Q_j^1, Q_j^2, ..., Q_j^l)$. For two distinct groups G and G', Q and Q' represents G and G' respectively. We define $G \prec_g G'$ *iff* $Q \prec Q'$.

In this paper, we study two kinds of aggregate function. The first one is strictly monotone, which means $f(Q_j^1, Q_j^2, ..., Q_j^l) > f(Q_j^{1'}, Q_j^{2'}, ..., Q_j^{l'})$ if $Q_j^i \geq Q_j^{i'}$ for every $i \in [1, l]$ and $\exists k$ such that $Q_j^k > Q_j^{k'}$, where $1 \leq k \leq l$. For the strictly monotone function, we study SUM in this paper. We also investigate aggregate functions that are not strictly monotone such as MAX and MIN. Fig. 2 shows the dominance relations under different aggregate functions.



	Points			SUM	MAX	MIN
G	Q^2(0,6)	Q^3(4,2)	Q^4(4,4)	(8,12)	(4,6)	(0,2)
G'	Q^3(4,2)	Q^4(4,4)	Q^5(0,5)	(8,11)	(4,5)	(0,2)
	Dominance Relation			$G \prec_g G'$	$G \prec_g G'$	$G = G'$

Figure 2: Dominance relations under different aggregate functions

Based on the Definition 1 or Definition 2, skyline group is defined as follows:

Definition 3. **(GSkyline)** The l-point GSkyline consists of groups with l points that are not dominated by any other groups of the same size.

We define the problem of top-k skyline groups query in the following. To facilitate the presentation, we define a function $score(G)$ that counts the number of the points dominated by group G. Then we have:

$$score(G) = |\{Q \in D - G | \exists Q' \in G \wedge Q' \prec Q\}|$$

For instance, if $G = \{Q^2, Q^4\}$, $score(G) = 2$.

Definition 4. **(k-SGQ)** Top-k skyline groups query retrieves the set $S_K \subseteq GSkyline$ of k skyline groups with highest score values. Then we have:

$$\forall G \in S_K, \forall G' \in (GSkyline - S_K) \rightarrow score(G) \geq score(G')$$

Obviously, k-SGQ can be applied to find top-k skyline groups based on both Definition 1 and Definition 2.

2.2 Related Work

The most related works with regard to the concept of skyline groups queries are [3, 4, 2, 6, 8, 9]. [3, 4, 2, 8, 9] investigate the skyline groups query based on Definition 2 and Liu et al. [6] investigate the problem based on Definition 1. However, the output sizes of both definitions are large, which is a potential limitation of skyline group operator. To solve this, we propose an efficient algorithm to select top-k skyline groups.

The most related works to our k-SGQ are [5] and [8]. [5] proposes a top-k representative skyline points query. It aims to compute a set of k skyline points such that the total number of points dominated by one of the k skyline points is maximized. Obviously, it is inherently different from our problem. Moreover, since a skyline group may consist of both skyline points and non-skyline points. Therefore, the techniques proposed in [5] are not applicable to our problem. [8] proposes an algorithm to find top-k combinatorial skyline. In their work, a combinatorial skyline is a skyline group based on Definition 2. They rank skyline groups based on a predefined preferred attribute order. It only reports groups whose aggregate values for a certain attribute are the highest. Obviously, the problem proposed in [8] is also inherently different from our problem. Moreover, the ranking method proposed in [8] is not applicable to Definition 1, because a group cannot be represented by a point based on this definition. Therefore, [8] is orthogonal to our problem.

To the best of our knowledge, we are the first to address the problem of finding top-k skyline groups that dominate the maximum number of points.

3. COMPUTING TOP-*K* GSKYLINE

The brute-force method to compute k-SGQ is to enumerate all skyline groups and count the number of points dominated by each group, then select the best k skyline groups. For each skyline group we need $O(l \times n)$ time complexity to count the points dominated by the group. Let $|SG|$ denote the size of skyline groups, then time complexity of selecting best k groups is $O(|SG| \times log\, k)$. Therefore, the overall time complexity is $O(l \times n \times |SG| \times log\, k)$. Obviously, the brute-force method incurs high computation overhead.

3.1 The *k*-SGQ Algorithm

Let *Skyline* denote the set of points in the skyline. *Skyline* is an accompanying result when computing *GSkyline* [6, 9]. We summarize frequently used notions in Table 1.

LEMMA 1. *For Definition 1 and strictly monotone aggregate functions under Definition 2, if $G \in GSkyline$ and $G = \{Q^1, Q^2, ..., Q^l\}$, then for each $Q^i \in G$, we have $Q^i \in Skyline$ or $\exists Q^j \in G$ and $Q^j \in Skyline \rightarrow Q^j \prec Q^i$.*

PROOF. We prove by contradiction. Assume that $Q^j \prec Q^i$ *and* $Q^j \in Skyline$, if $Q^j \notin G$, we can use Q^j to replace Q^i in G, the new group is denoted as G'.

CASE 1. *For the Definition 1, since $Q^j \prec Q^i$ and all the other points are the same, then $G' \prec_g G$ which contradicts $G \in GSkyline$.*

CASE 2. *For a strictly monotone aggregate function f under Definition 2, since $Q^j \prec Q^i$, we assume that $Q_t^j > Q_t^i$. Then we have $f(Q_t^1, ..., Q_t^i, ..., Q_t^l) < f(Q_t^1, ..., Q_t^j, ..., Q_t^l)$. On other dimensions, we have $f(Q_{t'}^1, ..., Q_{t'}^i, ..., Q_{t'}^l) \leq f(Q_{t'}^1, ..., Q_{t'}^j, ..., Q_{t'}^l)$. Therefore, $G' \prec_g G$, which contradicts $G \in GSkyline$.*

Therefore, for Definition 1 and strictly monotone aggregate functions under Definition 2, if $G \in GSkyline$, then for each $Q^i \in G$, we can get that $Q^i \in Skyline$ or $\exists Q^j \in G$ *and* $Q^j \in Skyline \rightarrow Q^j \prec Q^i$. $\square$

LEMMA 2. *For MAX and MIN under Definition 2, if $\exists Q^i \in G$ $(G \in GSkyline)$ and Q^i is dominated by at least*

Table 1: The Summary of Notations

Notation	Description
D	A d-demonical data set
d	Number of dimensions
n	Number of points in D
Q^i	The i^{th} point in D
Q^i_j	The value on the j^{th} dimension of Q^i
$\prec$	Preference/dominance relation
Skyline	The skyline of data set D
l	Size of a group
score(G)	Number of points dominated by G

k points, then either (1) $\exists Q^j \in G$ and $Q^j \prec Q^i$ or (2) it is safe to prune G from GSkyline.

Proof. Obviously, it is possible to have a point $Q^j \in G$ *and* $Q^j \prec Q^i$. In this situation we have $score(G) = score(G \setminus \{Q^i\})$.

In the second situation, all points dominate Q^i are not in G. If $Q^j \prec Q^i$, we use Q^j to replace Q^i in G, the new group is denoted as G'. Since $Q^j \prec Q^i$ and all the other points are the same, then $MAX(G') \preceq MAX(G)$ and $MIN(G') \preceq MIN(G)$. As G is a skyline group under MAX and MIN, we have $MAX(G') = MAX(G)$ and $MIN(G') = MIN(G)$ which means that G' is also a skyline group under MAX and MIN.

Moreover, we have $socre(G') \geq score(G)$. Since Q^i is dominated by at least k points, we have at least k skyline groups whose *socres* are equal or greater than $score(G)$. Therefore, it is safe to prune G from *GSkyline*. □

Let $dom(Q)$ denote the set of points dominated by point Q, then $score(G) = |\bigcup_{Q \in G} dom(Q)|$. In order to compute $score(G)$ efficiently, we maintain a bit vector for each point in the group, then we can employ fast bit-wise operations for much more efficient score computation.

Definition 5. **([Q])** $[Q]$ denotes the bit vector of Q. $[Q]$ has the length of $|D|$ bits, with one bit corresponding to a point in D. If a point Q^j is dominated by Q then the j^{th} bit is set to 1. Otherwise, the bit is set to 0.

Based on Definition 5, $score(G)$ equals the number of "1" in $[Q^1]||[Q^2]||...||[Q^l]$ ($G = \{Q^1, Q^2, ..., Q^l\}$). For instance in Fig. 1, $[Q^2] = 00001$, $[Q^4] = 00100$. If $G = \{Q^2, Q^4\}$, $score(G)$ equals the number of "1" in $[Q^2]||[Q^4] = 00101$. Therefore, $score(G) = 2$.

Based on Lemma 1 and Lemma 2, we do not need to compute $[Q]$ for every point in the data set. We use $(k-1)$-skyband [7] to denote the set of points that are dominated by at most $k-1$ points in a data set.

Lemma 3. *For Definition 1 and strictly monotone aggregate functions under Definition 2, based on Lemma 1, we know that if $Q \in G$ and $Q \notin Skyline$, then Q has zero contribution to $score(G)$. Thus $[Q]$ is modified as follows:*

$$[Q] = \begin{cases} [Q], Q \in Skyline \\ 0...0, \quad\quad Others \end{cases}$$

For MAX and MIN, we modify $[Q]$ in the following. Because if $Q \notin (k-1)$-skyband then either Q has zero contribution to $score(G)$ or it is safe to prune a candidate group containing Q, thus all points outside of the $(k-1)$-skyband will not affect the result of top-k skyline groups query.

$$[Q] = \begin{cases} [Q], Q \in (k-1)\text{-}skyband \\ 0...0, \quad\quad\quad\quad Others \end{cases}$$

Algorithm 1: *The k-SGQ algorithm*

```
Input  : $GSkyline$, $k$;
Output: the result set $S_K$ of $k$-SGQ on $GSkyline$
1  begin
2      $PQ \leftarrow \emptyset$; /* PQ is a priority queue sorting groups in the
           ascending order of their scores                                  */
3      $\tau \leftarrow -1$; /* $\tau$ is a threshold used for pruning        */
4      if the aggregate function is strictly monotone or under
       Definition 1 then
5          Compute the bit vectors for points in the Skyline;
6      if the aggregate function is $MAX$ or $MIN$ then
7          Compute the bit vectors for points in the
           $(k-1)$-skyband;
8      for each group $G$ in GSkyline do
9          if $MaxScore(G) > \tau$ then
10             if $score(G) > \tau$ then
11                 $PQ.push(G)$;
12             if $|PQ| > k$ then
13                 $PQ.pop()$;
14                 $\tau \leftarrow PQ.top().score$;
15     return $PQ$;
```

Definition 6. **(MaxScore)** *MaxScore* denotes the upper bound of *score*. $MaxScore(G) = \sum_{Q \in G} |dom(Q)|$.

$$|dom(Q)| = \begin{cases} 0, & [Q] = 0...0 \\ number\ of\ \text{"1"}\ in\ [Q], & [Q] \neq 0...0 \end{cases}$$

Obviously, $MaxScore(G) \geq score(G)$.

Lemma 4. *Let S_C be a candidate set containing k skyline groups and τ be the smallest score for all groups in S_C. For a specified group $G' \in GSkyline$ with $MaxScore(G') \leq \tau$, it can be safely pruned away as it cannot be an actual answer group for k-SGQ.*

Based on the above discussion, we propose Algorithm 1 to compute k-SGQ. In Algorithm 1 we maintain a priority queue of k skyline groups. Line 3 sets a threshold used for pruning. Then we compute the bit vectors for points in the *Skyline* or $(k-1)$-skyband based on the skyline group definition and aggregate functions. Line 9 utilizes Lemma 4 to prune candidate groups, Line 10 utilizes bit-wise operations to compute $score(G)$ for candidate groups.

3.2 Time Complexity Analysis

The time complexity of computing bit vectors for points in the *Skyline* and $(k-1)$-skyband is $O(|Skyline| \times n)$ and $O(|(k-1)\text{-skyband}| \times n)$ respectively. For each group G in *GSkyline*, we need $l-1$ bit-wise operations to get $score(G)$. The time complexity of updating the priority queue is $O(log\ k)$. Therefore, the time complexity of Algorithm 1 under Definition 1 and strictly monotone functions under Definition 2 is $O(|Skyline| \times n + (l-1) \times |SG| \times log\ k)$. The time complexity of Algorithm 1 for MAX and MIN is $O(|(k-1)\text{-skyband}| \times n + (l-1) \times |SG| \times log\ k)$. Obviously, the time complexity of Algorithm 1 is far less than the time complexity of the brute-force method.

4. EXPERIMENTAL EVALUATION

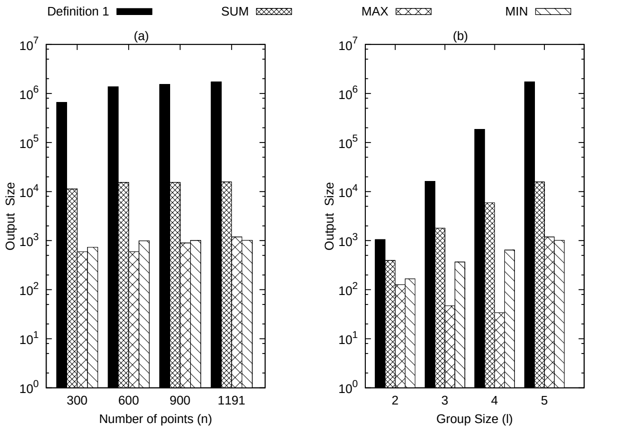


Figure 3: Output size of skyline groups based on Definition 1 and Definition 2 of varying n and l

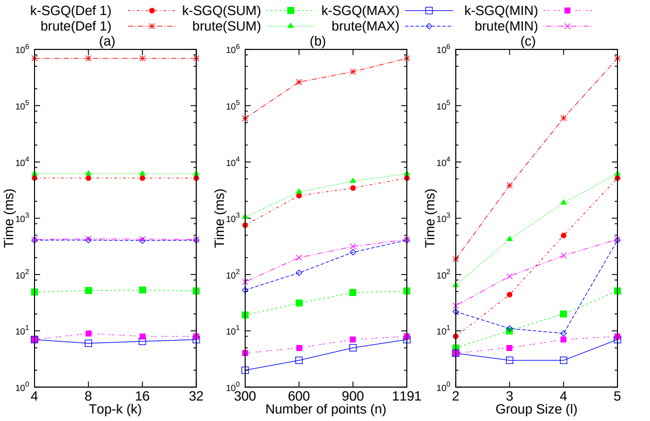


Figure 4: Performance of k-SGQ algorithm and brute-force method under different settings

In this section, we conduct extensive experiments to evaluate the run-time performance of Algorithm 1 under different settings. All our experiments are carried out on the same machine with 64GB memory and dual eight-core Intel Xeon E7-4820 processors clocked at 2.0Ghz. All algorithms are implemented in *C++*.

We continue to use the real data set adopted in [6]. The data set contains 1191 NBA players who are league leaders of playoffs. The data was extracted from http://stats.nba.com/leaders/alltime/?ls=iref:nba:gnav on 11/01/2016. Each player has five attributes that measure the player's performance. Those attributes are Points (PTS), Rebounds (REB), Assists (AST), Steals (STL) and Blocks (BLK).

We experiment with different settings, including number of best skyline groups k, number of points n and number of points in a group l. The settings of all these parameters are summarized in Table 2, where the default values are shown in bold. In every set of experiments, we only change one parameter, with the rest set to their defaults.

We compute skyline groups based on both Definition 1 and Definition 2. For Definition 2, we experiment with SUM, MAX and MIN. The output sizes of skyline groups are shown in Fig. 3. We can see that the output sizes based on both definitions are too large to make quick selections. Thus it is not trivial to design a top-k algorithm.

In Fig. 4 we present the performance of k-SGQ algorithm under different settings. We evaluate the performance of applying k-SGQ algorithm to find top-k skyline groups based on Definition 1 and Definition 2. From the three subfigures in Fig. 4 we can see that k-SGQ algorithm is efficient for both definitions under different settings. Therefore, our algorithm can be applied to all existing skyline group operators. Moreover, compared to the brute-force method, k-SGQ algorithm is much faster. For instance, there are 1720610 skyline groups generated from the NBA data set based on Definition 1, k-SGQ only needs 5 seconds to compute top-32 skyline groups while brute-force method needs 693 seconds. The experimental results show that for Definition 1 and SUM, k-SGQ algorithm is about average 134× and 121× speedup over the brute-force method respectively. For MAX and MIN, k-SGQ algorithm is about average 33× and 45× faster than the brute-force method respectively. Therefore, k-SGQ algorithm is efficient under different settings and can be applied to compute top-k skyline groups for all existing skyline group operators.

Table 2: Parameter Ranges and Default Values

Parameter	Range
k	4,8,16, **32**
n	300,600,900, **1191**
l	2,3,4, **5**

5. CONCLUSIONS

In this paper, we introduce a new and useful type of query, top-k skyline groups queries. The existing techniques cannot be applied to solve k-SGQ. We propose an efficient algorithm with several powerful pruning strategies. Moreover, we conduct extensive experiments to validate the efficiency our algorithm. Experimental results show that our algorithm reaches high performance under different settings.

6. ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China under Grant No.61502511, No.61572514, and No.61502513.

7. REFERENCES

[1] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *Proc. ICDE*, pages 421–430. IEEE, 2001.
[2] Y.-C. Chung, I.-F. Su, and C. Lee. Efficient computation of combinatorial skyline queries. *Information Systems*, 38(3):369–387, 2013.
[3] H. Im and S. Park. Group skyline computation. *Information Sciences*, 188:151–169, 2012.
[4] C. Li, N. Zhang, N. Hassan, S. Rajasekaran, and G. Das. On skyline groups. In *Proc. CIKM*, pages 2119–2123. ACM, 2012.
[5] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting stars: The k most representative skyline operator. In *Proc. ICDE*, pages 86–95. IEEE, 2007.
[6] J. Liu, L. Xiong, J. Pei, J. Luo, and H. Zhang. Finding pareto optimal groups: group-based skyline. *Proc. VLDB*, 8(13):2086–2097, 2015.
[7] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *TODS*, 30(1):41–82, 2005.
[8] I. F. Su, Y. C. Chung, and C. Lee. Top- k combinatorial skyline queries. In *Proc. DASFAA*, pages 79–93, 2010.
[9] N. Zhang, C. Li, N. Hassan, S. Rajasekaran, and G. Das. On skyline groups. *TKDE*, 26(4):942–956, 2014.

Towards sequenced semantics for evolving graphs

Vera Zaychik Moffitt
Drexel University
zaychik@drexel.edu

Julia Stoyanovich*
Drexel University
stoyanovich@drexel.edu

ABSTRACT

The research community has adopted a sequence of snapshots as the logical representation of evolving graphs — graphs that change over time and whose history of evolution we want to preserve for analysis. This paper argues that the snapshot sequence model of evolving graphs is insufficient for representation and analysis of a wide range of networks. Instead, we propose to use the interval model with sequenced semantics. In this model nodes and edges are associated with their validity intervals, and operations adhere to the properties of snapshot reducibility, extended snapshot reducibility, and change preservation. We show the advantages of adopting this model for evolving graphs and lay the groundwork for an evolving graph query language with sequenced semantics. We also discuss several challenges of efficiently supporting sequenced semantics in a distributed setting.

1. INTRODUCTION

Evolving graphs are used to represent a wide range of phenomena, including the Web, social networks, communication and transportation networks, interaction networks, metabolism pathways, and many others. Researchers study graph evolution rate and mechanisms, impact of specific events on further evolution, spatial and spatio-temporal patterns, and how graph properties change over time.

The dominant logical model for evolving graphs over the past 20 years has been a sequence of static graphs, termed *snapshots*. This model is a graph-specific adaptation of the *point-based temporal model* [19], and it introduces a semantic ambiguity that has been well studied in the temporal relational databases literature [2]: if an entity (graph, vertex or edge) with the same attributes exists in two consecutive snapshots, does it represent the same fact or two different facts? What does it mean for an entity to change?

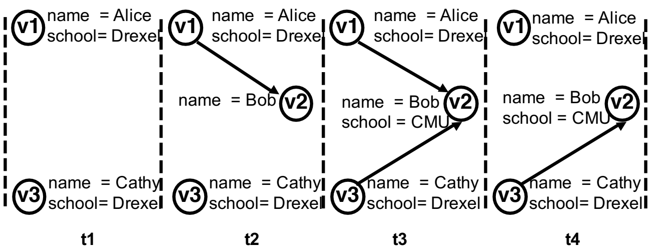


Figure 1: A social network as a snapshot sequence.

Figure 1 shows an example of an evolving social network, in which vertices represent people, while edges represent interactions between them such as likes and conversations. In this example, did Alice and Bob have two conversations over the time period $[t1, t4)$ or one long one? Did Alice undergo any changes during this time? Which user was the most active in this network, as defined by the number of distinct interactions? What is the rate of change of this network? We cannot answer these questions without additional information in a point-based model. Suppose that Alice held a temporary position at Drexel at time $t1$ and transferred to a permanent one at time $t2$. This information cannot be represented in the point-based model. Suppose that Alice and Bob had two short interactions, while Cathy and Bob had one longer one. The point-based model cannot distinguish between these two cases.

This kind of semantic ambiguity affects several graph operations, most notably aggregation and retrieval of change history, and, as a result, local (confined to specific entity or subset of entities) and global (whole-graph) temporal queries that are useful for evolving graph analysis.

In a point-based model [19] each entity is time-stamped with its validity time. For practical reasons, intervals are often used as syntactic abbreviations for sets of points. To use intervals in a time-stamped model, we coalesce, i.e., merge value-equivalent tuples over overlapping and adjacent time points [1]. Importantly, the use of intervals to represent a sequence of value-equivalent time-adjacent snapshots is not semantically equivalent to a model with *sequenced semantics*, where entities are time-stamped with intervals that have meaning. A work-around to avoid coalescing tuples that represent different facts is to add attributes to entities, in order to distinguish between changes and non-changes. For example, we can add position title to the vertex Alice to state that Alice changed jobs at time $t2$, and add a conversation id to each edge to designate distinct conversations. Unfortunately, this solution is ad-hoc rather than general and does not hold up over time, as discussed in [2].

*This work was supported in part by NSF Grants No. 1464327 and 1539856, and BSF Grant No. 2014391.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.41

We contend that a snapshot sequence model of evolving graphs is insufficient for representation of a wide range of networks and propose instead to use the interval model with sequenced semantics. Facts in our model correspond to graph vertices and edges, rather than to graph snapshots in their entirety (as in Figure 1). This representational choice is orthogonal to the issue of point-based vs. sequenced semantics, but has an important advantage. Many evolving graph queries include temporal predicates over vertices or edges, e.g., compute a subgraph containing only vertices that persist for at least a year. Such queries cannot be evaluated directly over a sequence of snapshots.

In Section 2 we briefly survey existing models and summarize relevant work in temporal databases. We then propose a new model in Section 3. In Section 4 we discuss the challenges of efficient computation under sequenced semantics in a distributed environment. We conclude with future research directions in Section 5.

2. RELATED WORK

Evolving graph models. While temporal models in the relational literature are very mature, the same cannot be said about the evolving graphs literature. Evolving graph models differ in what time stamp they use (point or interval stamping), what top-level entities they model (graphs or sets of nodes and edges), whether they represent topology only or attributes or weights as well, and what types of evolution are allowed. All evolving graph models require node identity, and thus edge identity as well, to persist across time. See [20] for a survey of evolving graph models.

The first mention of evolving graphs that we are aware of is by Harary and Gupta [6] who informally proposed to model the evolution as a sequence of static graphs. This model has been predominant in the research literature ([4, 7, 14] and many others), with various restrictions on the kinds of changes that can take place during graph evolution. For example, Khurana and Deshpande [7] use this model with the restriction that a node, once removed, cannot reappear. In [4] and [14] there is no notion of time, only a sequence of graphs. It is important to note that we are talking about the logical model of the evolving graphs, rather than a physical representation. For example, Semertzidis et al. [16] present a concrete representation of an evolving graph called VersionGraph that is similar to the logical model we propose here, yet their logical model is still a sequence of snapshots.

The advantages of the snapshot sequence model are that (a) it is simple and (b) if snapshots are obtained by periodic sampling, which is a very common approach, it accurately represents the states of the graph at the sampled points without making assertions about unknown times. For example, the WWW is so large that it is impossible to create a fully accurate snapshot that represents any moment in time. An important limitation of this model, in addition to the semantic ambiguity on what constitutes a change, is that it forces a specific time granularity, whereas open-closed time intervals can be broken down into any desired level of granularity.

Temporal relational models. The question of semantics of temporal data has been thoroughly explored in the relational temporal database community. Böhlen et al. [2] defined point and sequenced models, and showed that the difference between the models lies in the properties of the operators, and not in the use of intervals as representational devices. With this foundation, Dignös et al. [3] defined sequenced semantics, with properties of snapshot reducibility, extended snapshot reducibility, and change preservation. Snapshot reducibility means that a temporal operator produces the same result as an equivalent non-temporal operator over corresponding snapshots. Extended snapshot reducibility allows references to timestamps in the operators by propagating them as data. Point semantics has both of these properties as well.

The third property, change preservation, is unique to sequenced semantics. It states that operators only merge contiguous time points of a result if they have the same lineage. As shown in [3], all three properties can be guaranteed through the use of the normalize and align operators on non-temporal relations, extended with an explicit time attribute.

As we as a community move to more and more sophisticated analyses of evolving graphs, we need to adopt the state of the art in temporal databases.

3. DATA MODEL

We now describe the *logical* representation of an evolving graph, called a TGraph. A TGraph represents a single graph, and models evolution of its topology and of vertex and edge attributes.

Following the SQL:2011 standard [8], a period (or interval) $\boldsymbol{p} = [s, e)$ represents a discrete set of time instances, starting from and including the start time s, continuing to but excluding the end time e. Time instances contained within the period have limited precision, and the time domain has total order. In the rest of this paper we use the terms interval and timestamp interchangeably.

A TGraph is represented with four temporal SQL relations [1], and uses sequenced semantics [3], associating a fact (existence of a vertex or edge, and an assignment of a value to a vertex or edge attribute) with an interval.

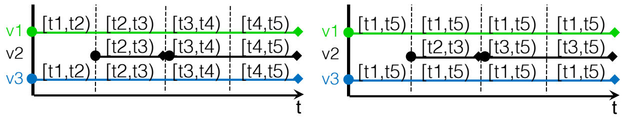
A snapshot of a temporal relation R, denoted $\tau_c(R)$ is the state of R at time point c.

We use the property graph model [15] to represent vertex and edge attributes: each vertex and edge during period $\boldsymbol{p}$ is associated with a (possibly empty) *set* of properties, and each property is represented by a key-value pair. Property values are not restricted to be of atomic types, and may, e.g., be sets, maps or tuples.

We now give a formal definition of a TGraph, which builds on the model of [12] and is adjusted to support sequenced semantics.

DEFINITION 3.1 (TGRAPH). *A TGraph is a pair* $\mathsf{T} = (\mathsf{V}, \mathsf{E})$. *$\mathsf{V}$ is a valid-time temporal SQL relation with schema $\mathsf{V}(\underline{v}, \underline{\boldsymbol{p}})$ that associates a vertex with the time period during which it is present. E is a valid-time temporal SQL relation with schema $\mathsf{E}(\underline{v_1}, \underline{v_2}, \underline{\boldsymbol{p}})$, connecting pairs of vertices from V. T optionally includes vertex and edge attribute relations $\mathsf{A}^\mathsf{V}(\underline{v}, \underline{\boldsymbol{p}}, a)$ and $\mathsf{A}^\mathsf{E}(\underline{v_1}, \underline{v_2}, \underline{\boldsymbol{p}}, a)$, where a is a nested attribute consisting of key-value property pairs. Relations of T must meet the following requirements:*

- **R1: Unique vertices/edges** *In every snapshot $\tau_c(\mathsf{V})$ and $\tau_c(\mathsf{E})$ a vertex/edge exists at most once.*
- **R2: Unique attribute values** *In every snapshot $\tau_c(\mathsf{A}^\mathsf{V})$ and $\tau_c(\mathsf{A}^\mathsf{E})$, a vertex/edge is associated with at most one attribute (which is itself a set of key-value pairs representing properties).*



(a) with fragments (b) with full timestamps

Figure 2: Vertices from Figure 1 split in 4 partitions.

R3: Referential integrity *In every snapshot $\tau_c(\mathsf{T})$, foreign key constraints hold from $\tau_c(E)$ (on both v_1 and v_2) and $\tau_c(\mathsf{A}^{\mathsf{V}})$ to $\tau_c(V)$, and from $\tau_c(\mathsf{A}^{\mathsf{E}})$ to $\tau_c(E)$.*

Requirements **R1, R2, R3** guarantee soundness of the TGraph data structure, ensuring that every snapshot of a TGraph is a valid graph. Graphs may be directed or undirected. For undirected graphs we choose a canonical representation of an edge, with $v_1 \leq v_2$ (self-loops are allowed). At most two edges can exist between any two vertices at any time point, one in each direction.

Definition 3.1 presents a *logical* data structure that admits different physical representations, including, e.g., a columnar representation (each property in a separate relation, supporting different change rates), by a hash-based representation of [18], or in some other way. The logical model also allows for distributed storage in HDFS.

4. SEQUENCED SEMANTICS IN A DISTRIBUTED ENVIRONMENT

Many interesting static graphs are so large that they necessitate a distributed approach, as evidenced by the plethora of works on Pregel-style computation and graph partitioning [10]. In this section we discuss the challenges inherent in supporting three properties of sequenced semantics — snapshot reducibility, extended snapshot reducibility, and change preservation — in a distributed environment.

Snapshot reducibility. Evolving graphs can be partitioned among the available machines using time locality. Following convention, we refer to the operator that can produce such partitioning as a *splitter*. The splitter places each tuple (vertex or edge) into one or more partitions based on its timestamp. The goal of the splitter is to form partitions that are balanced, i.e., have approximately the same number of items, under the assumption that most operations can be executed locally at each partition. Recall that snapshot reducibility requires a temporal operator to produce the same result as if it were evaluated over each snapshot. Validity period of a tuple that spans more than one temporal partition is split, and the tuple is replicated across partitions. This increases the overall size of the relation, but all operations can now be carried out within each partition. See Figure 2a for a simple example of the V relation being split into four temporal partitions.

For the purposes of illustration, consider the temporal subgraph operation, a generalization of subgraph matching for non-temporal graphs [12]. Temporal vertex-subgraph $\mathsf{sub}_v^T(q_v^t, \mathsf{T}) = \mathsf{T}'(\mathsf{V}', \mathsf{E}', \mathsf{A}^{\mathsf{V}'}, \mathsf{A}^{\mathsf{E}'})$ computes an induced subgraph of T, with vertices defined by the temporal conjunctive query q_v^t. Note that this is a subgraph query, and so $\mathsf{V}' \subseteq^T \mathsf{V}$. Observe that we can carry out the subgraph operation with non-temporal predicates, e.g., name='Alice', at each partition individually, without any cross-partition communication.

The question of optimal splitting has been addressed by Le et al. [9], who demonstrated that a temporal relation can be efficiently split into k buckets in cases of both internal memory and external memory, and guarantee optimality of the solution. This method requires a sequential scan of the relation to compute an index called the stabbing count array. How to make this method more efficient in a distributed environment is an open question.

The subgraph operation requires co-partitioning of graph relations to enforce referential integrity on edges. A number of alternatives for co-partitioning present themselves, as the vertex, edge and attribute relations are not guaranteed to have the same splitters due to different evolution rates. Typically, vertices are co-partitioned with edges in the non-temporal case [5], and this likely is most efficient with evolving graphs as well.

Extended snapshot reducibility. Snapshot reducibility can be guaranteed in the distributed setting, as shown above for a subgraph query without temporal predicates. In general, a subgraph query q_v^t may use any of the constituent relations of T, and may explicitly reference temporal information in compliance with the extended snapshot reducibility property of sequenced semantics. Refer back to Figure 2a and assume time granularity of years. If we perform the subgraph operation, selecting vertices that persist for longer than 2 years, over the split then we will get no matches. However, the original relation contains two matches – only Bob does not meet the predicate. To support extended snapshot reducibility over a split relation, during partitioning tuples should be placed into their partitions with their full original timestamps. Incidentally, this is what Le at al. describe in their work on optimal splitters [9]. Figure 2b shows relation V split in the same four partitions with this approach.

Change preservation. Change preservation property requires that derived tuples should only be coalesced if they share lineage. To support this property, normalize and align operators are used [3]. The normalize operator splits each tuple in the input relation w.r.t. a group of tuples such that each timestamp fragment is either fully contained or disjoint with every timestamp in the group. The align operator splits each tuple w.r.t. a group of tuples such that each timestamp fragment is either an intersection with one of the tuples in the group or is not covered by any tuple in a group.

The normalize operator splits each tuple w.r.t. to a group defined by the operation. For example, consider the attribute-based node creation operation on graphs [12], an operation similar to aggregation on temporal relations. This operation allows the user to generate a TGraph in which vertices correspond to disjoint groups of vertices in the input that agree on the values of all grouping attributes. For instance, $\mathsf{node}_a^T(\mathsf{school}, \mathsf{T})$ will compute a vertex for each value of A^{V}.a.school. While the group defined for each tuple (distinct value of school) spans temporal partitions, only tuples within the same partition overlap. Thus, the normalize and align operations can be carried out locally at each partition.

An important challenge to address is how to efficiently support aggregation over temporal windows in a distributed setting. This operation requires cross-partition communication, which impacts the cost model, requiring a generalization of the approach of Le et al. [9].

Partitioning of evolving graphs. Large evolving graphs present additional challenges compared to static graphs and

Table 1: Connected components and PageRank with different temporal partitioning, seconds.

(a) wiki-talk

width	k	CC	PR
8	23	382	1,086
16	12	328	1,037
bal.	16	351	720
bal.	24	408	375

(b) nGrams

width	k	CC	PR
8	26	1,324	2,867
16	13	856	1,873
bal.	3	321	740
bal.	16	422	519

temporal relations alone. Each graph snapshot may be too large to fit into a single partition. This necessitates partitioning graphs by both time and structure. Miao et al. [11] have demonstrated within their ImmortalGraph system that different locality, structural or temporal, is more appropriate for different graph queries. However, their results do not directly translate to the distributed environment. Miao et al. showed that spatial locality provides better performance than temporal locality in global point queries, i.e., queries that compute over a snapshot corresponding to a particular time point. In a distributed setting we do not expect these results to hold, since communication costs generally dominate the overall performance, and partitioning by time alone will guarantee that a snapshot is distributed among the lowest number of partitions.

Global range queries such as change in graph centrality over time, on the other hand, are computed over multiple snapshots and their performance depends on the method of computation. We can utilize temporal locality and compute on each snapshot independently. Assuming that each partition fits one or more snapshots, the maximum number of snapshots across all partitions will determine the overall performance. With structural locality we can distribute edges across the partitions using any of the already proposed partitioning approaches such as range- and hash-based [17] or EdgePartition2D (E2D).

We explored the effectiveness of partitioning strategies in graphs that undergo changes in topology over time, and found that structural partitioning such as in ImmortalGraph is effective only when graph topology changes very little [13]. We found that a hybrid approach that combines temporal and structural locality is promising. We are currently investigating methods for selecting a partitioning strategy that provides the best overall performance.

Preliminary experiments. We conducted some preliminary experiments to see the effect of temporal partitioning on distributed execution of analytics, which present one of the heaviest computational workloads. PageRank and Connected components analytics were executed on the wiki-talk[1] and nGrams[2] datasets.

Wiki-talk contains 179 time periods. nGrams contains over 400, but we used the first 208. Both datasets exhibit strong skew, with few edges at the start of the datasets and increasing by several orders of magnitude towards the end. We compared equi-width and equi-depth temporal partitioning, using 8 and 16 consecutive intervals for equi-width, and using offline optimal split of edges with varying number of splitters k. Each dataset was partitioned first temporally, and then spatially using Edge2D partitioning. Table 1 shows that equi-depth partitioning is superior to equi-width in all cases but one. However, the number of splitters is key in obtaining good results. We are currently investigating this phenomenon further.

5. CONCLUSION

We have argued that modeling evolving graphs using snapshot sequences presents semantic difficulties. As an alternative, we proposed a vertex-edge model with sequence semantics and discussed several challenges of supporting this model in a distributed setting. We are currently working on implementing this model in Apache Spark, and exploring the performance of different physical representations and partition strategies.

6. REFERENCES

[1] M. H. Böhlen et al. Coalescing in temporal databases. In *VLDB*, 1996.

[2] M. H. Böhlen et al. Point versus interval-based temporal data models. In *ICDE*, 1998.

[3] A. Dignös et al. Temporal alignment. In *SIGMOD*, 2012.

[4] A. Fard et al. Towards efficient query processing on massive time-evolving graphs. In *IEEE CollaborateCom*, 2012.

[5] J. E. Gonzalez et al. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[6] F. Harary and G. Gupta. Dynamic graph models. *Mathematical and Computer Modelling*, 25(7), 1997.

[7] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *ICDE*, 2013.

[8] K. G. Kulkarni and J. Michels. Temporal features in SQL: 2011. *SIGMOD Record*, 41(3), 2012.

[9] W. Le et al. Optimal splitters for temporal and multi-version databases. In *SIGMOD*, 2013.

[10] R. R. McCune et al. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM CSUR*, 1(1), 2015.

[11] Y. Miao et al. ImmortalGraph: A system for storage and analysis of temporal graphs. *ACM TOS*, 11(3), 2015.

[12] V. Z. Moffitt and J. Stoyanovich. Querying evolving graphs with portal. Dec. 2016. arXiv:1602.00773.

[13] V. Z. Moffitt and J. Stoyanovich. Towards a distributed infrastructure for evolving graph analytics. In *TempWeb*, 2016.

[14] C. Ren et al. On querying historical evolving graph sequences. *PVLDB*, 4(11), 2011.

[15] I. Robinson, J. Webber, and E. Eifrem. *Graph databases.* O'Reilly Media, Inc., 2013.

[16] K. Semertzidis et al. TimeReach: Historical reachability queries on evolving graphs. In *EDBT*, 2015.

[17] J. Seo et al. Distributed socialite: A datalog-based language for largescale graph analysis. *PVLDB*, 6(14), 2013.

[18] W. Sun et al. SQLGraph: An efficient relational-based property graph store. In *SIGMOD*, 2015.

[19] D. Toman. Point-stamped temporal models. In *Encyclopedia of Database Systems.* 2009.

[20] A. Zaki et al. Comprehensive survey on dynamic graph models. *IJACSA*, 7(2), 2016.

[1] http://dx.doi.org/10.5281/zenodo.49561

[2] http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

Crowdsourcing Strategies for Text Creation Tasks

Ria Mae Borromeo
Keio University
Yokohama, Japan
ria@db.ics.keio.ac.jp

Maha Alsayasneh, Sihem Amer-Yahia, Vincent Leroy
Univ. Grenoble Alpes, CNRS, LIG,
F-38000 Grenoble, France
firstname.lastname@imag.fr

ABSTRACT

We examine deployment strategies for text translation and text summarization tasks. We formalize a deployment strategy along three dimensions: *work structure*, *workforce organization*, and *work style*. Work structure can be either simultaneous or sequential, workforce organization independent or collaborative, and work style either crowd-only or hybrid. We use Amazon Mechanical Turk to evaluate the cost, latency, and quality of various deployment strategies. We asses our strategies for different scenarios: short/long text, presence/absence of an outline, and popular/unpopular topics. Our findings serve as a basis to automate the deployment of text creation tasks.

Keywords

Crowdsourcing; Text Creation; Deployment Strategies;

1. INTRODUCTION

Crowdsourcing has been applied to all kinds of tasks ranging from the simplest such as image categorization to the most sophisticated such as creating elaborate text. Although several automatic solutions have been designed for text creation, this task remains difficult for machines as it involves a level of abstraction and creativity that only humans are capable of. That is particularly true for translation and summarization where original texts of varying length and complexity need to be understood and processed. In this paper, we examine how hybrid deployment strategies that combine the power of algorithms with the creativity of humans can improve the quality of produced text, as well as the cost and latency of tasks. To the best of our knowledge, our work is the first to explore the effectiveness of hybrid deployment strategies for crowdsourced text creation.

We are interested in two text creation tasks: translation and summarization. It has been shown that for text translation, letting workers edit text and correct each others' mistakes in a sequential manner, produces higher quality translations than in the case where workers generate independent translations simultaneously [1]. It has also been shown that automatic methods are not very good at summarizing and merging sentences to generate high-quality summaries [11]. We hence propose to study different deployment strategies. A deployment strategy is a plan on how to carry out a task. It is a combination of three dimensions: *work structure*, *workforce organization*, and *work style*. Work structure refers to how a task is deployed among workers, which can either be *simultaneous* or *sequential*. Workforce organization refers to how workers are organized to complete a task, which can either be *independent* or *collaborative*. Work style distinguishes a *hybrid* approach, where a task is completed by both algorithms and humans, from a *crowd-only* approach, where a task is solely carried out by humans. Table 1 shows 6 deployment strategies that combine those dimensions.

The idea of combining humans and machines for task completion has been explored in a variety of domains ranging from databases to machine learning [3, 4, 5, 7, 9, 10, 12]. Our focus is on the evaluation of how our strategies affect cost, latency, and quality of output text. For translation, in addition to work structure, workforce organization, and work style, we pay attention to the properties of the text that is being translated or summarized and study the impact of text length. For summarization, we study the quality of summaries in the presence and absence of a suggested outline, and for topics of varying popularity.

The paper is organized as follows. Our tasks and deployment strategies are given in Section 2. Our experiments are presented in Section 3. We conclude and discuss perspectives raised by this work in Section 4.

2. TASK DEPLOYMENT

2.1 Translation

We examine two types of translation tasks: full document and short text translation. In the first case, the original text is a speech by President Obama entitled “Giving Every Student an Opportunity to Learn Through Computer Science for All.” It consists of 35 sentences and 10 paragraphs. The target language is French. In the second case, the original text is a poem in Arabic, “When You Decide to Leave” by Mahmoud Darwish, with 4 sentences. The target languages are English and French.

2.2 Summarization

We chose movie reviews and soccer games to be summarized into free-text, structured, or personalized summaries. A free-text summary is generic and has no specific struc-

Strategy	Description
Sequential-Independent-Hybrid (**SEQ-IND-HYB**)	An initial output is generated automatically then it is sent to one worker at a time for improvement. The final result is a single output.
Sequential-Independent-CrowdOnly (**SEQ-IND-CRO**)	An initial output is completed by a worker then it is sent to one worker at a time to improve it. The final result is a single output.
Simultaneous-Independent-Hybrid (**SIM-IND-HYB**)	An initial output is generated automatically then sent to several independent workers for improvement. The best output is chosen after an evaluation.
Simultaneous-Independent-CrowdOnly (**SIM-IND-CRO**)	Several outputs are created simultaneously by independent workers. The best output is chosen after an evaluation.
Simultaneous-Collaborative-Hybrid (**SIM-COL-HYB**)	An initial output is generated automatically then sent to one group of workers who collaborate to improve it.
Sequential-Collaborative-CrowdOnly (**SIM-COL-CRO**)	One output is created by one group of workers together.

Table 1: Deployment Strategies

ture while the structured and personalized ones are based on a given outline. A structured summary, however, puts more emphasis on the organization of text, while in a personalized summary the content is given primary importance. The choice of movies and soccer allows us to control topic popularity.

For movies, we used IMDb datasets, and chose "The Imitation Game," "2012," and "The Count of Monte Cristo" as they respectively satisfy the following characteristics: popular with high ratings, popular with low ratings, and not popular. For each movie, we selected five reviews with 7 to 10 sentences each, to be summarized in at most 7 sentences.

For soccer, we asked to summarize game statistics into 14 sentences. We chose two games that were recently held at La Liga-Spain 2016: one between two popular teams, Barcelona and Granada, and the other between less popular teams, Rayo Vallecano and Levante.

2.3 Deployment Strategies

Figure 1 illustrates all strategies for the translation task. For instance, SEQ-IND-HYB first generates an initial translation from English to French using Google Translate, then it asks three workers to improve the translation one after the other. In addition to the original text and task instructions, a requester must consider the following: the number of workers to recruit for the task and the result quality requirement, which are affected by time and budget constraints. For example, in translating Obama's speech, a requester may expect the highest possible quality that three workers can achieve within no particular time and without budget restrictions.

Since we want the highest possible quality, we evaluate every response received. The evaluation may be done by experts, by algorithms [8], or by the crowd [2].

3. VALIDATION

In this section, we report the setup and the results of experiments we performed to evaluate our proposed deployment strategies. We deployed our tasks on Amazon Mechanical Turk (AMT). The list of required skills was provided at the beginning of each task. In the case of hybrid strategies, we used Google Translate to obtain machine translations and MEAD[1] to obtain automatic summaries.

We observed how our strategies affect the cost, latency and result quality. We calculated the *cost* by taking the sum of all the payments to workers for all the HITs posted to carry out a strategy. We asked experts to rate *quality* of each text output using a 5-pt Likert scale (1 - very poor, 2 - poor 3 - barely acceptable, 4 - good, 5 - very good) using the following criteria: spelling, syntax, semantic coherence, and adequation to the original text. The *latency* was derived by adding the amount of time it took for a worker or group of workers to complete each task in a given strategy. Table 2 summarizes the comparisons that we performed.

3.1 Translation

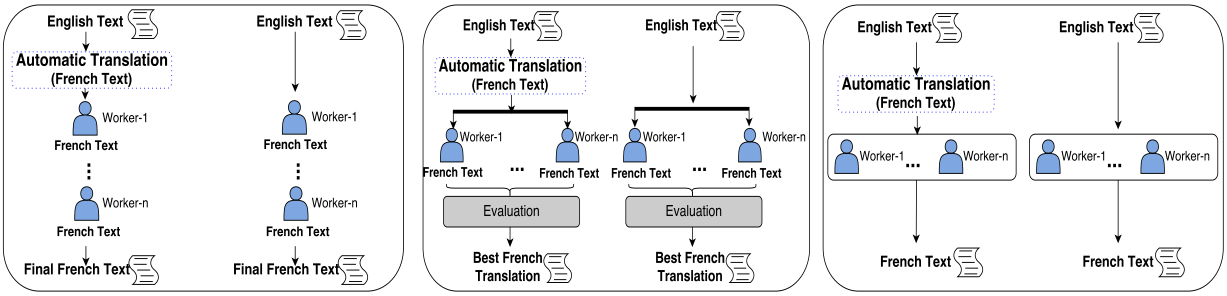
All strategies were considered to translate Obama's speech. For Darwish's poem, we only report results for *simultaneous* work structure and *independent* workforce organization (Figure 1b). Sequential strategies were not useful since the text is short. Similarly, for collaborative strategies, the time and effort of recruiting workers outweighs their benefit for short text.

Setup. Figure 1a shows how we implemented *sequential independent* strategies for translation tasks. In the case of a *hybrid* work style, we first obtained an automatic translation of the original text to the target language. It was then improved by three different workers one after the other. To improve a translation, we published a Human Intelligence Task (HIT) that instructs a worker to enhance an automatically produced translation. For every response, we asked an expert to rate the improved translation. When the rating was good enough, we asked the next worker to enhance the current translation. Otherwise, we asked another worker to enhance the initial translation until we received an acceptable translation. For a *crowd-only* work style, we first published a HIT that requests a translation of the original text from scratch. After receiving an initial translation, we asked two more workers to improve the translation iteratively.

As shown in Figure 1b, for *simultaneous independent* strategies, we posted a HIT requesting three workers to translate text simultaneously. For the *hybrid* work style, workers improved an initial machine translation, while in the *crowd-only* case, they translated the original text from scratch. After receiving all three answers, we asked an expert to select the best one.

For *simultaneous collaborative* strategies (Figure 1c), we needed workers to collaborate to create (*crowd-only*) or improve (*hybrid*) a translation. We deployed these tasks by

[1] *http://www.summarization.com/mead/*



(a) SEQ-IND-HYB & SEQ-IND-CRO (b) SIM-IND-HYB & SIM-IND-CRO (c) SIM-COL-HYB & SIM-COL-CRO

Figure 1: Translation strategies

Workforce Organization (IND vs COL)	**Work Structure (SIM vs SEQ)**	**Work Style (HYB vs CRO)**
SIM-IND-HYB vs. SIM-COL-HYB	SIM-IND-HYB vs. SEQ-IND-HYB	SEQ-IND-CRO vs. SEQ-IND-HYB
SIM-IND-CRO vs. SIM-COL-CRO	SIM-IND-CRO vs. SEQ-IND-CRO	SIM-IND-CRO vs. SIM-IND-HYB
		SIM-COL-HYB vs. SIM-COL-CRO

Table 2: Comparison Scenarios

posting a HIT that explains to workers the task requirements and asks them if they are willing to work on the task with other workers. After that, we invited at least two workers to use Google Docs to collaborate on the translation.

We based our incentives on the pricing scheme in [13] that paid $0.10 (US dollars) per sentence. For the independent tasks, we paid $3.50/HIT for each translation from scratch and $1.75 for each translation improvement HIT. For the collaborative tasks, we paid $1.16 per worker for the translation from scratch HIT and $0.58 per worker for the translation improvement HIT.

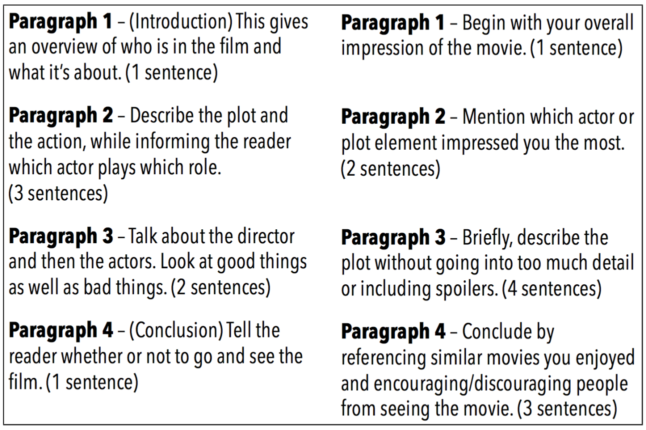
Findings. We find that letting workers collaborate as a group has a positive impact on the behavior of workers, which also contributes to raising translation quality. Another advantage of collaboration is a much lower cost, while latency only slightly increases. For translating long text, a hybrid work style combined with a sequential work structure are best (SEQ-IND-HYB). For short text, however, a simultaneous work structure is more appropriate, and both hybrid and crowd-only work styles perform well (SIM-IND-HYB and SIM-IND-CRO).

3.2 Summarization

It has been shown that providing a narrative outline improves text summaries [6]. To verify this finding for movies and soccer games, we crowdsourced summaries with various deployment strategies in the presence and the absence of a proposed summary outline for topics of varying popularity. The summarization tasks were deployed using the same strategies as in Figures 1a, 1b, and 1c with movie reviews or game statistics as input, and a summary text as output. For the *hybrid* work style, we first obtained an automatic summary using MEAD and gave it to workers for improvements. In the case of a *crowd-only* work style, workers were instead provided an outline to follow when producing a summary. This work was then performed with different workforce organizations and work structures, similar to translation tasks.

Movie Setup. We selected movies with different ratings and popularity. In addition to a structured outline that we provided, we asked three different workers to propose an outline that conforms to their expectations (personalized). Figure 2 shows two example outlines. On the left side is our proposed outline. On the right side is one that a worker suggested. One can see that ours is generic and covers the main aspects of a movie while that of the worker is more specific. We deployed all strategies for the movie "The Imitation Game" to obtain free-text summaries as well as personalized summaries. The two other movies were summarized with *crowd-only* work styles, using a structured summary. The incentives we provided for the independent creation of free-text summaries are as follows: $5.00 for each written from scratch, $1.25 for its 1^{st} improvement and $0.62 for the 2^{nd}; $2.50 for the 1^{st} improvement of an automatically generated summary, $0.62 for the 2^{nd}, and $0.31 for the 3^{rd}. For collaborative tasks, we paid each worker $1.16 to create a summary from scratch and $0.58 each to improve a summary. For the structured and personalized summaries of movie reviews, we paid $0.70 for the task of coming up with an outline and for creation and improvement tasks.

Movie Findings. We find that workers produce better summaries when given an outline that serves as a template. This finding reinforces previous results in narrative theory that show an increase in emotional worker engagement, and the likelihood of workers sharing those summaries when narrative templates are used to produce them [6]. However, there is a fine line between providing outlines that are general and outlines that ask for specific content requiring workers to spend extra time finding that content. We also observe that a hybrid work style that provides workers with an automatically generated initial summary helps workers structure their thoughts. Among our proposed strategies, we found that SEQ-IND-HYB is best for free-text summaries while SEQ-IND-CRO is best for structured ones. Finally, we find that summarizing reviews for a popular movie does not guarantee a high-quality outcome.



Paragraph 1 – (Introduction) This gives an overview of who is in the film and what it's about. (1 sentence)	**Paragraph 1** – Begin with your overall impression of the movie. (1 sentence)
Paragraph 2 – Describe the plot and the action, while informing the reader which actor plays which role. (3 sentences)	**Paragraph 2** – Mention which actor or plot element impressed you the most. (2 sentences)
Paragraph 3 – Talk about the director and then the actors. Look at good things as well as bad things. (2 sentences)	**Paragraph 3** – Briefly, describe the plot without going into too much detail or including spoilers. (4 sentences)
Paragraph 4 – (Conclusion) Tell the reader whether or not to go and see the film. (1 sentence)	**Paragraph 4** – Conclude by referencing similar movies you enjoyed and encouraging/discouraging people from seeing the movie. (3 sentences)

Figure 2: Two Summarization Outlines

Soccer Setup. We sought to verify how the popularity of a team and the deployment strategy affect the results. We requested summaries for two games: one between two popular teams and another between less popular teams. The soccer games were summarized with *crowd-only* work styles, using a structured summary. To create structured summaries for soccer games, we paid \$1.40 and to improve a summary, we also paid \$0.70.

Soccer Findings. We observed that the summaries created for popular teams were completed faster and were of higher quality compared to the less popular game. We also noticed that workers prefer working *independently* and *sequentially* (SEQ-IND-CRO), as they tend to disagree a lot on this topic, which makes *collaborating* difficult.

4. SUMMARY AND PERSPECTIVES

The main takeaway is that humans have an aversion to long text and to the effort of creating text from scratch (case of full document translation). They are however better than machines at sequentially improving automatically translated text, or at creating text based on outlines (case of summarizing movie reviews). For short text, providing an initial machine translation does not help.

The popularity of an event affects the quality of obtained summaries (case of soccer games). Its recency impacts the speed at which workers respond. Also, for tasks requiring creativity, and when the input text is short (case of the short poem), humans are best. The same is true when guidelines are provided for text creation tasks (case of summary outlines). However, when those guidelines are too specific, the resulting quality drops as it becomes necessary to focus on finding answers to specific questions (case of personalized summary outlines).

Our findings can serve as a basis for the development of automatic task deployment text creation. In particular, we would like to design an environment that lets requesters interact with suggested deployment strategies and refine them as tasks are completed. This requester-in-the-loop perspective will provide more transparency in crowdsourcing.

5. REFERENCES

[1] P. André, R. E. Kraut, and A. Kittur. Effects of simultaneous and sequential work structures on distributed collaborative interdependent tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 139–148. ACM, 2014.

[2] C. Callison-Burch. Fast, cheap, and creative: evaluating translation quality using amazon's mechanical turk. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1-Volume 1*, pages 286–295. Association for Computational Linguistics, 2009.

[3] J. Fan, M. Lu, B. C. Ooi, W.-C. Tan, and M. Zhang. A hybrid machine-crowdsourcing system for matching web tables. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 976–987. IEEE, 2014.

[4] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 61–72. ACM, 2011.

[5] D. Haas, J. Ansel, L. Gu, and A. Marcus. Argonaut: macrotask crowdsourcing for complex data processing. *Proceedings of the VLDB Endowment*, 8(12):1642–1653, 2015.

[6] J. Kim and A. Monroy-Hernandez. Storia: Summarizing social media content based on narrative theory using crowdsourcing. *arXiv preprint arXiv:1509.03026*, 2015.

[7] G. Li, J. Wang, Y. Zheng, and M. Franklin. Crowdsourced data management: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, PP(99), 2016.

[8] N.-Q. Luong, L. Besacier, and B. Lecouteux. Towards accurate predictors of word quality for machine translation: Lessons learned on french–english and english–spanish systems. *Data & Knowledge Engineering*, 96:32–42, 2015.

[9] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Crowdsourced databases: Query processing with people. In *5th Biennial Conference on Innovative Data Systems Research*. CIDR, 2011.

[10] A. G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: declarative crowdsourcing. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1203–1212. ACM, 2012.

[11] G. Salton, A. Singhal, M. Mitra, and C. Buckley. Automatic text structuring and summarization. *Information Processing & Management*, 33(2):193–207, 1997.

[12] H. Wu, H. Sun, Y. Fang, K. Hu, Y. Xie, Y. Song, and X. Liu. Combining machine learning and crowdsourcing for better understanding commodity reviews. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[13] O. F. Zaidan and C. Callison-Burch. Crowdsourcing translation: Professional quality from non-professionals. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 1220–1229. Association for Computational Linguistics, 2011.

Poster Paper

Hybrid LSH: Faster Near Neighbors Reporting in High-dimensional Space

Ninh Pham
University of Copenhagen
Denmark
pham@di.ku.dk

ABSTRACT

We study the r-near neighbors reporting problem (rNNR) (or *spherical range reporting*), i.e., reporting *all* points in a high-dimensional point set S that lie within a radius r of a given query point. This problem has played building block roles in finding near-duplicate web pages, solving k-diverse near neighbor search and content-based image retrieval problems. Our approach builds upon the locality-sensitive hashing (LSH) framework due to its appealing asymptotic sublinear query time for near neighbor search problems in high-dimensional space. A bottleneck of the traditional LSH scheme for solving rNNR is that its performance is sensitive to data and query-dependent parameters. On data sets whose data distributions have diverse local density patterns, LSH with inappropriate tuning parameters can sometimes be outperformed by a simple linear search.

In this paper, we introduce a hybrid search strategy between LSH-based search and linear search for rNNR in high-dimensional space. By integrating an auxiliary data structure into LSH hash tables, we can efficiently estimate the computational cost of LSH-based search for a given query regardless of the data distribution. This means that we are able to choose the appropriate search strategy between LSH-based search and linear search to achieve better performance. Moreover, the integrated data structure is time efficient and fits well with many recent state-of-the-art LSH-based approaches. Our experiments on real-world data sets show that the hybrid search approach outperforms (or is comparable to) both LSH-based search and linear search for a wide range of search radii and data distributions in high-dimensional space.

1. INTRODUCTION

We study the r-near neighbors reporting problem (rNNR) (or *spherical range reporting*) [2, 5]: *Given a d-dimensional point set S of size n, reporting* all *points in S that lie within a radius r of a given query point.* This problem has played building block roles in finding near-duplicate web pages [11], solving k-diverse near neighbor search [1] and content-based image retrieval problems [15]. Recent theoretical work [2, 3] conjectures that solving rNNR exactly in time truly sublinear in n seems to demand space exponential in d, which is an example of the phenomenon "curse of dimensionality".

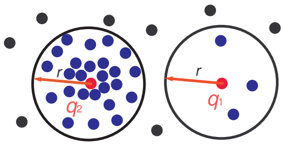


Figure 1: An example of LSH bottleneck. Given a radius r, LSH works efficiently with the query q_1 on sparse area, since it will report just a few points. However, LSH is worse than linear search with the "hard" query q_2 on dense area. Since the output size of q_2 is nearly the data set size and many points are very close to q_2, duplicates show up in most hash tables and the cost of removing duplicates will be the computational bottleneck.

Since exact solutions of rNNR generally degrade as dimensionality increases, we investigate an *approximate* variant of rNNR. That is, given a parameter $0 < \delta < 1$, we allow the algorithm to return each point in S that lie within a radius r of the query point with probability $1 - \delta$. Our approach builds upon on the *locality-sensitive hashing* (LSH) [4, 12], one of the most widely used solution for near neighbor search problems. In a nutshell, LSH hashes near points into the same bucket with good probability, and increases the gap of collision probability between near and far points. It typically needs to use multiple hash tables to obtain probabilistic guarantees. Search candidates are *distinct* data points that are hashed into the same bucket as the query in hash tables.

Since its first introduction, several LSH schemes [6, 7, 8, 10, 13, 14] have been proposed for a wide range of metric distances in high-dimensional space. However, a bottleneck of using LSH for solving rNNR is that its performance is sensitive to the parameters which depend on the distance distribution between data points and query points. Such parameters are hard to tune on data sets whose data distributions have diverse *local* density patterns. Figure 1 shows an illustration of this bottleneck.

In practice, LSH needs to use significant space (i.e., hundreds of hash tables) [10] or the multi-probe approach [13] which examines several "close" buckets in a hash table. In other words, the number of examined buckets needs to be sufficiently large to obtain high accuracy. In turn, the cost of removing duplicates (i.e., points colliding with the query in several hash tables) turns out to be the computational bottleneck when there are many points close to the query. This observation has been shown on the Webspam dataset in the experiment section even with very small radii.

In this work, we study a hybrid search strategy between LSH-based search and linear search for rNNR in an arbitrary high-dimensional space and distance measure that allows LSH. By integrating the so-called HyperLogLog data structures [9] into LSH hash tables, we can quickly and accurately estimate the output size and derive the computational cost of LSH-based search for a given query point regardless of the data distribution. In other words, we are able to choose the appropriate search strategy between LSH-based search and linear search to achieve better performance (i.e., running time and recall ratio). Moreover, the proposed solution can be adapted to many recent state-of-the-art LSH-based approaches [2, 13, 14]. Our experiments on real-world datasets demonstrate that the proposed hybrid search outperforms (or is comparable to) both LSH-based search and linear search for a wide range of search radii and data distributions in high-dimensional space.

2. BACKGROUND AND PRELIMINARIES

Problem setting. Our problem, r-near neighbor reporting under any distance measure, is defined as follows:

Definition 1. *(r-near neighbor reporting or rNNR) Given a set $S \subset \mathbf{R}^d$, $|S| = n$, a distance function f, and parameters $r > 0$, $\delta > 0$, construct a data structure that, given any query $\boldsymbol{q} \in \mathbf{R}^d$, return* each *point $\boldsymbol{x} \in S$ where $f(\boldsymbol{x}, \boldsymbol{q}) \leq r$ with probability $1 - \delta$.*

We call this the "exact" rNNR problem in case $\delta = 0$, otherwise it is the "approximate" variant.

Locality-sensitive hashing (LSH). LSH can be used for solving approximate rNNR in high-dimensional space because its running time is usually better than linear search with appropriate tuning parameters [4].

Definition 2. (Indyk and Motwani [12]) *Fix a distance function $f : \mathbf{R}^d \times \mathbf{R}^d \rightarrow \mathbf{R}$. For positive reals r, c, p_1, p_2, and $p_1 > p_2$, $c > 1$, a family of functions $\mathcal{H}$ is (r, cr, p_1, p_2)-*sensitive *if for uniformly chosen $h \in \mathcal{H}$ and all $\boldsymbol{x}, \boldsymbol{y} \in \mathbf{R}^d$:*

- *If $f(\boldsymbol{x}, \boldsymbol{y}) \leq r$ then $\mathbf{Pr}\left[h(\boldsymbol{x}) = h(\boldsymbol{y})\right] \geq p_1$;*
- *If $f(\boldsymbol{x}, \boldsymbol{y}) \geq cr$ then $\mathbf{Pr}\left[h(\boldsymbol{x}) = h(\boldsymbol{y})\right] \leq p_2$.*

Given an LSH family $\mathcal{H}$, the classic LSH algorithm constructs L hash tables by hashing data points using L hash functions g_j, $j = 1, \ldots, L$, by setting $g_j = \left(h_j^1, \ldots, h_j^k\right)$, where h_j^i, $i = 1, \ldots, k$, are chosen randomly from the LSH family $\mathcal{H}$. Concatenating k such random hash functions h_j^i increases the gap of collision probability between near points and far points. To process a query $\boldsymbol{q}$, one needs to get a candidate set by retrieving all points from the bucket $g_j(\boldsymbol{q})$ in the jth hash table, $j = 1, \ldots, L$. Each *distinct* point $\boldsymbol{x}$ in the candidate set is reported if $f(\boldsymbol{x}, \boldsymbol{q}) \leq r$.

For the approximate rNNR, a near neighbor has to be reported with a probability at least $1 - \delta$. Hence, one can fix the number of hash tables, L, and set the value k as a function of L and δ. A simple computation indicates that $k = \left\lceil \log\left(1 - \delta^{1/L}\right) / \log p_1 \right\rceil$ leads to good performance[1]. Note that our parameter setting is different from the standard setting $k = \log n, L = n^\rho$, where $\rho = \log p_1 / \log p_2$ [12], since we focus on reporting *every* r-near neighbor.

Although LSH-based algorithm can efficiently solve rNNR problem, it might run in $\mathcal{O}\left(nL\right)$ time in the worst case, see Figure 1 as an example. Tuning appropriate parameters k, L for a given dataset whose data distribution has diverse local density patterns remains a tedious process.

HyperLogLog (HLL) for count-distinct problem. While counting the exact number of distinct elements in a data stream is simple with space linear to the cardinality, approximating such the cardinality using limited memory is an important problem with broad industrial applications. Among efficient algorithms for the problem, HyperLogLog (HLL) [9] constitutes the state-of-the-art (i.e., a near-optimal probabilistic algorithm) when there is no prior estimate of the cardinality. This means that it achieves a superior accuracy for a given fixed amount of memory over other techniques.

HLL builds an array M of m zero registers. For an element i, it generates a random *integer* pair $\{m_i, v_i\}$ where $m_i \sim \texttt{Uniform}([m])$ indicates a position in M, and $v_i \sim \texttt{Geometric}(1/2)$ is an update value. The array M updates the value at the position m_i by $\max\left(M[m_i], v_i\right)$. After processing all elements, the cardinality estimator of the stream is $\theta_m m^2 \left(\sum_{j=1}^m 2^{-M[j]}\right)$, where θ_m is a constant to correct the bias. HLL works optimally with *distributed* data streams since we can merge several HLLs by collecting register values and applying component-wise a `max` operation. The relative error of HLL is $1.04/\sqrt{m}$. More details of the theoretical analysis and a practical version of HLL can be seen in [9].

3. ALGORITHM

This section describes our novel hybrid search strategy which interchanges LSH-based search and linear search for solving rNNR. We first present a simple but accurate computational cost model to measure the performance of LSH-based search. By constructing an HLL data structure in each bucket of hash tables, we are able to estimate the computational cost of LSH-based search, and then identify the condition whether LSH-based search or linear search is used.

3.1 Computational Cost Model

For each query, LSH-based search needs to process following operations: (1) Step S1: Compute hash functions to identify the bucket of query in L hash tables, (2) Step S2: Look up in each hash table the points of the same bucket of query, and merge them together for removing duplicate to form a candidate set, and (3) Step S3: Compute the distance between candidates and the query to report near neighbor points. Typically, the cost of S1 is very small and dominated by the cost of S2 and S3, which significantly depend on the distance distribution between the query and data points.

To process Step S2, one typically uses a hash table or a bitvector of n bits to store non-duplicate entries. The cost of such techniques is proportional to the total number of collisions (*#collisions*) encountered in L hash tables, which can be directly computed by simply storing the bucket size. The cost of S3 is clearly proportional to the candidate set size (*candSize*). The total cost of LSH-based search is composed of the cost of S2 and S3, as formalized in Equation (1).

Given α as the average cost of removing a duplicate, and β as the cost of a distance computation, we formalize the

[1] This is a practical setting used in E2LSH package (http://www.mit.edu/∼andoni/LSH/)

total cost of LSH-based search and linear search as follows:

$$\textbf{LSHCost} = \alpha \cdot \#\mathit{collisions} + \beta \cdot \mathit{candSize} \quad (1)$$

$$\textbf{LinearCost} = \beta \cdot n \quad (2)$$

Given such constants α, β, we can compute exactly **LinearCost**, but we need *candSize* for computing **LSHCost**. By constructing an HLL data structure for each bucket, we can derive the HLL of the candidate set. Therefore, we can accurately approximate *candSize*, and then estimate the **LSHCost**. In turn, we can compare **LinearCost** and **LSHCost** in order to *interchange* LSH-based search with linear search to achieve better performance.

3.2 Hybrid Search Strategy

We construct an HLL for each bucket when building LSH hash tables, as shown in Algorithm 1. Given a query $\boldsymbol{q}$, we view point indexes hashed in the buckets $g_1(\boldsymbol{q}), \cdots, g_L(\boldsymbol{q})$ as L partitions of a data stream. We will estimate the number of distinct elements of such data stream, which is the *candSize* in Equation (1). By estimating **LSHCost** and comparing it to **LinearCost**, we can identify the suitable search strategy, as shown in Algorithm 2.

Algorithm 1 Construct LSH hash tables

```
Require: A point set $S$, and $L$ hash functions: $g_1, \ldots, g_L$
for each $\boldsymbol{x} \in S$ do
    for each hash table $T_i$ using hash function $g_i$ do
        Insert $\boldsymbol{x}$ into the bucket $g_i(\boldsymbol{x})$
        Update HyperLogLog of the bucket $g_i(\boldsymbol{x})$
    end for
end for
```

Algorithm 2 Hybrid search for r-NN

```
Require: A query point $\boldsymbol{q}$, and $L$ hash tables: $T_1, \ldots, T_L$
1: Get the size of the buckets $g_1(\boldsymbol{q}), \ldots, g_L(\boldsymbol{q})$ to compute $\#collisions$
2: Merge HLLs of the buckets $g_1(\boldsymbol{q}), \ldots, g_L(\boldsymbol{q})$ to estimate $candSize$
3: Estimate LSHCost using Equation (1), and compute LinearCost using Equation (2)
4: Choose LSH-based search if LSHCost < LinearCost; otherwise, use linear search
```

The time complexity analysis. Now, we analyze the complexity of the two algorithms. Algorithm 1 uses a space overhead due to the additional HLLs. For each bucket, an HLL needs $O(m)$ space where m is the number of registers of HLL, which governs the accuracy of the *candSize* estimate. In practice, we only need $m = 32 - 128$. This means that the space overhead of HLLs is usually smaller than large buckets (e.g., $\#points > m$). For small buckets (e.g., $\#points < m$), we might not need HLL, since we can update the merged HLL on demand at the query time. This trick can save the space overhead and improve the running time of the algorithm.

Algorithm 2 is more important since it governs the running time of the algorithm. Compared to the classic LSH-based search, the additional cost of the hybrid search approach is from merging L HLL data structures and estimating *candSize*, which takes $O(mL)$. Such cost is often smaller than (or comparable to) the cost of Step S1, i.e., hash functions computation on LSH families [6, 7, 8, 12]. In other words, the cost overhead caused by our hybrid search approach is little and dominated by the total search cost.

4. EXPERIMENT

We implemented algorithms in Python 3 and conducted experiments on an Intel Xeon Processor E5-1650 v3 with 64GB of RAM. We compared the performance of different search strategies, including hybrid search, LSH-based search, and linear search for reporting near neighbors on several metric distances allowing LSH. We used 4 real-world data sets: Corel Images[2] ($n = 68,040, d = 32$), CoverType[2] ($n = 581,012, d = 54$), Webspam[3] ($n = 350,000, d = 254$), and MNIST[3] ($n = 60,000, d = 780$). For each dataset, we randomly remove 100 points and use it as the query set, and report the average of 5 runs of algorithms on the query set.

For each metric distance, we use the corresponding LSH family. Particularly, we applied SimHash [7] to obtain 64-bit fingerprint vectors for MNIST and use bit sampling LSH [12] for Hamming distance. CoverType and Corel Images use random projection-based LSH [8] for L1 and L2 distances, respectively. Webspam uses SimHash [7] for cosine distance.

4.1 Efficiency of HyperLogLog

This subsection presents experiments to evaluate the efficiency of HLLs on estimating the candidate set size for a given query point. For HLL's parameter, we fix $m = 128$ to achieve a relative error at most 10% as suggested in [9]. For LSH's parameters, we fix $L = 50$ and set $k = \left\lceil \log(1 - \delta^{1/L}) / \log p_1 \right\rceil$, where $\delta = 10\%$ and p_1 is the collision probability for points within the radius r to the query. This setting is used for SimHash [7] and bit sampling LSH [12]. For random projection-based LSH [8] for L1 and L2 distances, in order to achieve $\delta = 10\%$, we have to adjust $k = 8, w = 4r$ and $k = 7, w = 2r$, respectively, where w is an additional parameter of such LSHs. We note that HLL estimation takes $O(mL)$ time, so this cost is almost constant when fixing m and L.

Table 1: Relative cost and error of HLLs

Dataset	Webspam	CoverType	Corel	MNIST
% Cost	1.31%	0.12%	3.18%	17.54%
% Error	5.99%	5.86%	6.74%	6.8%

Table 1 shows the average performance of HLL over 4 datasets for a small range of radii where LSH-based search significantly outperforms linear search. It is clear that the cost of HLL is very little, less than 4% of the total cost for the real-value data points. For MNIST, since the distance computation cost is very cheap due to binary representation, the cost of HLL is 17.54% of the total cost. However, since MNIST is very small ($n = 60000$), we can set $m = 32$ to reduce the cost to 4.4% without degrading the performance.

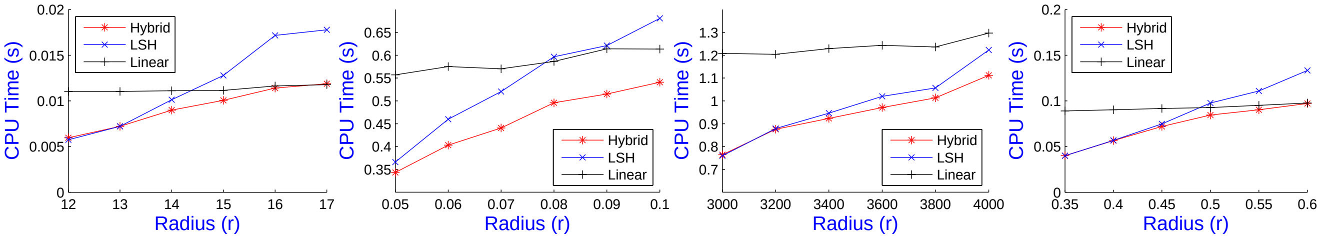
Regarding the accuracy, although theoretical analysis guarantees a relative error of 10%, the practical relative error is even much smaller, less than 7% with standard deviation around 5% for all datasets. The small overhead cost and high accuracy provided by HLL enables us to efficiently estimate the total cost of LSH-based search, see Equation (1), and identify the appropriate search strategy.

4.2 Efficiency of Hybrid Search

This subsection studies the performance of our proposed hybrid search strategy. To compare **LSHCost** and **LinearCost**, we need to identify the ratio β/α, which obviously depends on the implementation, the sparsity of the dataset

[2] https://archive.ics.uci.edu/ml/datasets/

[3] http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/



(a) MNIST with Hamming dist.

(b) Webspam with cosine dist.

(c) CoverType with L1 dist.

(d) Corel with L2 dist.

Figure 2: Comparison of CPU Time (s) for a query set between hybrid search (Hybrid), LSH-based search (LSH), and linear search (Linear) on 4 data sets using different metric distances.

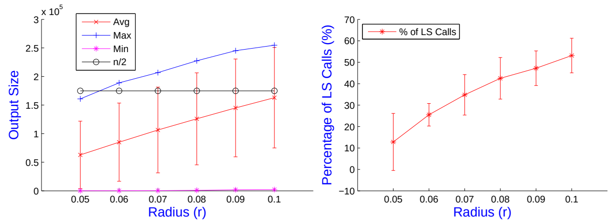


Figure 3: Left: Average, maximum, and minimum output size of queries; Right: Percentage of linear search (LS) calls used in hybrid search for Webspam.

and the used distance metric. We use a random set of 100 queries and 10,000 data points for choosing the ratio β/α as 10, 10, 6, 1 for Webspam, Covertype, Corel, and MNIST, respectively. We use the same setting as the previous section for LSH's and HLL's parameters.

Figure 2 shows the average running time in seconds of the 3 search strategies. For small r, LSH-based search and hybrid search are comparable, but superior to linear search since the output size of each query is rather small. When r increases, hybrid search gains substantial advantages by interchanging LSH-based search with linear search since there are more "hard" queries on the query set. It outperforms LSH-based search and eventually converges to linear search. Specifically, hybrid search provides superior performance compared to both LSH-based search and linear search on Webspam, as shown in Figure 2.b. This is due to the fact that Webspam has several "hard" queries for even very small radii ($r \leq 0.1$).

Figure 3 reveals that the output size varies significantly even with small r. The maximum output size is almost more than half of the point set size ($n/2$) whereas the minimum output size is very tiny. This means that Webspam has many "hard" queries, and therefore hybrid search gives superior average performance. The right figure confirms this observation by showing the average percentage of linear search calls for hybrid search. This amount is at least 10% at $r = 0.05$ and increases to approximate 50% at $r = 0.1$.

We note that hybrid search gives higher recall ratio than LSH-based search since it uses linear search for "hard" queries. Due to the limit of space, we do not report it here.

5. CONCLUSIONS

In this paper, we propose a hybrid search strategy for LSH on rNNR problem in high-dimensional space. By integrating an HyperLogLog data structure for each bucket, we can estimate the total cost of LSH-based search and choose the appropriate search strategy between LSH-based search and linear search to achieve better performance. Our experiments on real-world data sets demonstrate that the proposed approach outperforms (or is comparable to) both LSH-based search and linear search for a wide range of search radii and data distributions in high-dimensional space. We observed that our hybrid search fits well with the multi-probe LSH schemes [2, 13] and the covering LSH [14], which typically require a large number of probes. Applying hybrid search on these LSH schemes for rNNS will be our future work.

6. REFERENCES

[1] S. Abbar, S. Amer-Yahia, P. Indyk, and S. Mahabadi. Real-time recommendation of diverse related articles. In *WWW*, 2013.

[2] T. D. Ahle, M. Aumüller, and R. Pagh. High-dimensional spherical range reporting by output-sensitive multi-probing LSH. SODA'17.

[3] J. Alman and R. Williams. Probabilistic polynomials and hamming nearest neighbors. In *FOCS*, 2015.

[4] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, Jan. 2008.

[5] S. Arya, G. D. da Fonseca, and D. M. Mount. A unified approach to approximate proximity searching. In *ESA*, 2010.

[6] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *STOC*, 1998.

[7] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.

[8] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*, 2004.

[9] Éric Fusy, O. G, and F. Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AofA*, 2007.

[10] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[11] M. R. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*, 2006.

[12] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, 1998.

[13] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *VLDB*, 2007.

[14] R. Pagh. Locality-sensitive hashing without false negatives. In *SODA*, 2016.

[15] F. X. Yu, S. Kumar, Y. Gong, and S. Chang. Circulant binary embedding. In *ICML*, 2014.

SPST-Index: A Self-Pruning Splay Tree Index for Caching Database Cracking

Pedro Holanda
UFPR, Brazil
pttholanda@inf.ufpr.br

Eduardo Cunha de Almeida
UFPR, Brazil
eduardo@inf.ufpr.br

ABSTRACT

In database cracking, a database is physically self-organized into cracked partitions with cracker indices boosting the access to these partitions. The AVL Tree is the data structure of choice to implement cracker indices. However, it is particularly cache-inefficient for range queries, because the nodes accessed only for a few times (i.e, "Cold Data") and the most accessed ones (i.e, "Hot Data") are spread all over the index. In this paper, we present the Self-Pruning Splay Tree (SPST) data structure to index database cracking and reorganize "Hot Data" and "Cold Data" to boost the access to the cracked partitions. To every range query, the SPST rotates to the root the nodes pointing to the edges and to the middle value of the predicate interval. Eventually, the most accessed tree nodes remain close to the root improving CPU and cache activity. On the other hand, the least accessed tree nodes remain close to the leaves and are pruned to improve updates. Our experimental evaluation shows 37% more Instructions per Cycle and 75.9% less cache misses in L1 for lookup operations in the SPST compared to the AVL tree. Our data structure outperforms the AVL tree for lookups and maintenance costs in three major data access patterns: random, sequential and skewed. The SPST outperforms the AVL in 4% even in the worst case scenario with mixed workloads with lookups and batch updates.

Keywords

Database Cracking, Cracker Index, Splay Tree

1. INTRODUCTION

Database Cracking [6] presents a self-organizing database partitioning for column-oriented relational databases. It works by physically self-organizing database columns into partitions, called cracked pieces. The goal is to create cracked pieces for all accessed intervals of range queries. Cracker indices are created to keep track of these partitions.

An index is a data access method that typically stores a list of pointers to all disk blocks that contain records to the indexed column. The values in the index are ordered to make binary search possible. It is smaller than the data file itself, so searching the index using binary search is reasonably efficient [2]. In contrast to usual indices in the literature, the nodes of a cracker index do not point to all the disk blocks of a column. Instead, they point to the beginning of each cracked piece to boost access to an interval of values.

The current data structure implemented as cracker index is the self-balancing AVL Tree [1], where the height of the adjacent children subtrees of any node differ by at most one. If in a given moment their height differs by more than one, the tree is rebalanced by tree rotations. As the index is created by incoming queries, the index starts to be filled with pointers to data keeping the self-balancing property of the tree height. However, this property makes the AVL tree particularly cache-inefficient. The tree nodes accessed only for a few times (i.e, "Cold Data") and the most accessed ones (i.e, "Hot Data") are spread all over the index. Another concern lies in the index size as "Cold Data" are kept in the index. Eventually, the cracker index converges to a full index (i.e, all values indexed) with high administration costs for high-throughput updates.

In this paper, we present a data structure called Self-Pruning Splay Tree (SPST) to index database cracking and keep "Hot Data" close to the root of the tree. The SPST is based on binary search Splay trees with a self-adjusting property carried out by the *splaying* operation. *Splaying* consists of a sequence of rotations to move a node way up to the root of the tree. To every range query, our algorithm rotates the nodes pointing to the edges and to the middle value of the predicate interval. With "Hot Data" constantly rotated, they eventually remain close to the root. On the other hand, "Cold Data" are stored close to the leaves presenting the opportunity to prune them out of the index and improve maintenance and update costs.

This paper is organized, as follows: Section 2 discusses related work. Section 3 presents our Cracker Index followed by the experiments in Section 4 and we conclude in Section 5.

2. STANDARD DATABASE CRACKING AND RELATED WORK

There are two Database Cracking algorithms: *crack-in-two* and *crack-in-three* to split the columns into two and three partitions respectively. The first one is suited for one-sided range queries (e.g, $V_1 < A$) or two-sided range queries (e.g, $V_1 < A < V_2$) where each side accesses different cracked pieces. The second one is only for two-sided queries that access the same cracked piece. It starts with similar per-

Series ISSN: 2367-2005

10.5441/002/edbt.2017.44

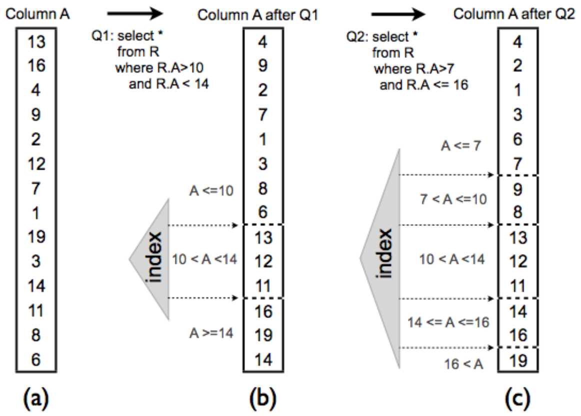


Figure 1: Database Cracking when executing two queries with different ranges [8]

formance of full column scan and overtime gets close to the performance of a full index.

Figure 1 depicts query Q_1 triggering the creation of the cracker column A_{ckr}, (i.e., initially a copy of column A) where the tuples are clustered in three pieces reflecting a *crack-in-three* iteration from the range predicate of Q_1. The result of Q_1 is then retrieved as a view on Piece 2 (i.e., indexing $10 < A < 14$). Later, query Q_2 requires a refinement of Pieces 1 and 3 (i.e., respectively indexing $A > 7$ and $A \leq 16$), splitting each in two new pieces resulted by a *crack-in-two* iteration.

There are many data structures in the literature to keep track of data partitions. In database cracking the AVL is the data structure of choice, but other self-balancing trees, like RedBlack or 2-3 trees, draw the same result. These trees have the property of keeping the height of the tree for self-balancing purposes. However, this property makes them cache-inefficient for range queries. The tree nodes accessed only for a few times and the most accessed ones are spread all over the tree.

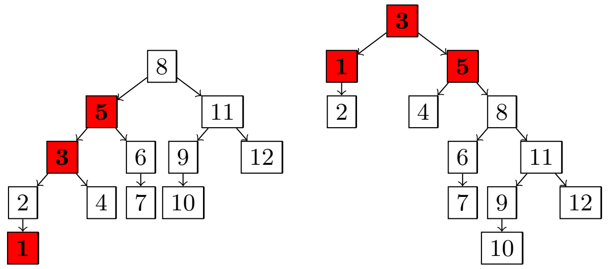
3. THE SPST-INDEX

Our contribution regards recognizing "hot data" to improve data access and recognizing "cold data" to prune unused data and boost updates.

3.1 Splaying

A Splay Tree [9] is a self-adjusting binary search tree that uses a splaying technique every time a node is Searched, Updated, Inserted or Deleted. *Splaying* consists of a sequence of rotations that moves a node to the root of the tree. Lookup, Insertion and Deletion take $O(\log n)$ time in the average and worst case scenarios, where n is the number of nodes in the Splay Tree. It clusters the most accessed nodes near the root of the tree. Therefore, the most frequent accessed nodes will be accessed faster. Since we are dealing with range queries, our goal is to splay the query range, instead of splaying only one node like the original splay tree. The self-adjustment algorithm in our data structure is straightforward: we first splay the leftmost node of the range, then the rightmost node and later the closest node to the middle.

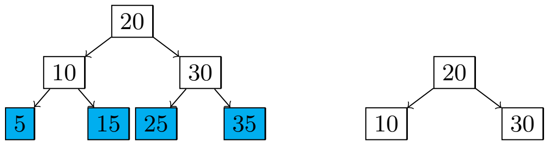
Let us consider for cracker index the SPST depicted by Figure 2. If a range query of $1 < A < 5$ is executed, the algorithm performs three operations: Splay (1), Splay (5) and Splay ($\lceil \frac{(1+5)}{2} \rceil$). Figure 2(b) depicts the resulting tree with nodes 1, 3 and 5 close to the root. In the SPST, the nodes remain close to the root as long as they are frequently accessed. In our index, the nodes pointing to the most accessed cracked pieces remain close to the root.



(a) The SPST before the incoming query $1 < A < 5$

(b) The SPST after Splaying nodes 1, 5, and 3

Figure 2: The SPST splaying the range $1 < A < 5$



(a) The SPST before pruning the leaves

(b) The SPST after pruning the leaves

Figure 3: The SPST with size $n = 7$ being pruned.

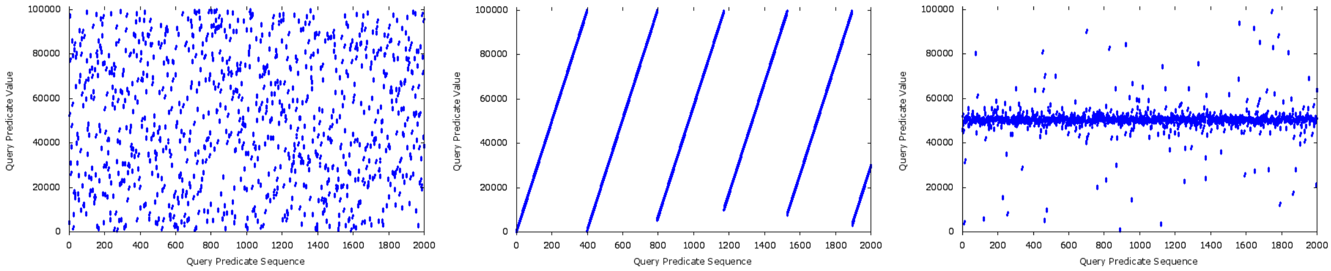
3.2 Pruning

Besides speeding up the access to hot data, another goal is to speed up updates and maintenance costs when rotating hot data. We assume that eventually the nodes stored at the leaves point to cold data. The maintenance strategy of our data structure is to prune the leaves. As we prune them, the update time is expected to shrink. The downside of pruning the tree is that the following queries can become slightly more expensive compared to the situation where we do not have any pruning at all. Our hypothesis is that we mitigate this cost with the gains in the update time. When we prune the leaves, the size of the index shrinks, in the best case, to $\lfloor \frac{n}{2} \rfloor$, where n is the number of nodes in the SPST.

Let us suppose the SPST index depicted by Figure 3(a). In this scenario the most frequent range is between 10 and 30. Let us suppose inserting the value 21 in the Cracker Column. To do this, we need to update the nodes 35, 30 and 25 respective pointers to the cracker column and merge at their respective cracker column pieces. Instead, we start pruning the leaves having as result the tree depicted by Figure 3(b). Then we only need to update the pointer to the cracked piece of node 30.

4. EXPERIMENTAL ANALYSIS

In this section, we discuss the results of our experimental evaluation of the SPST implemented as a cracker index. We divide this section in two subsections, the first one is related to the select operator where we performed the same



(a) Random (b) Sequential (c) Skewed

Figure 4: Workload Patterns

experimental protocol and ran the same lookup scenarios described in [8]. The second one is related to the update scenario where we performed the same experimental protocol and ran the same scenarios described in [4]. We implemented our data structure and performed all the experiments using the database cracking simulator[1] presented by [8]. We ran the experiments on a MacOS Sierra (10.12) machine with 2.2GHz quad-core Intel Core i7 processor (Turbo Boost up to 3.4GHz), 6MB shared L3 cache and 8 GB of RAM.

For the select operator, we focused our analysis on the accumulated index lookup time for querying and indexing, and the accumulated index update time. In particular, we analyzed the Instructions per Cycle (IPC) and the cache misses (L1/2/3). We consider as the best cache-efficient data structure the one with the highest IPC and lowest number of cache misses.

For the update operator, we considered two update scenarios: low frequency high volume updates (i.e, LFHV), and high frequency low volume updates (i.e, HFLV). In the first scenario after 1,000 queries a batch of 1,000 updates are executed. In the second scenario after 10 queries a batch of 10 updates are executed. The query pattern and the updates are both random. The SPST prunes itself always before a batch update if the previous queries present a standard deviation, for cracking time, lower than a defined threshold. We focused our analysis only on measurements that are affected by update and pruning (i.e, cracking time, index update time, cracker column shuffle time and pruning time).

We use an integer array with 10^8 uniformly distributed values. The workload size and the query selectivity is 1,000 and 1 for all experiments. All query predicates are of the form: $R.A \geq V_1$ AND $R.A < V_2$. We repeat the entire workload 5 times and take the average runtime of each query. We consider three different workloads depicted by Figure 4. For each workload, we graphically illustrate how a sequence of $1,000$ queries accesses the domain value of a single attribute. For each query, we plot the two edges of the interval (i.e., called "Query Predicate Sequence"). The random, sequential and skewed workloads are respectively depicted by Figures 4(a), 4(b), and 4(c). The skewed workload is generated by the zipf's law with α equals to 2.0.

4.1 Select Operator

Figure 5 depicts the accumulated index lookup and maintenance time for the query stream in the random, sequential and skewed workload. For random, the AVL Tree was faster than the SPST for the first 180 queries, because the random workload demanded a higher number of rotations in the SPST to settle down the range pattern close to the root. With more incoming queries the SPST started to leverage the cached nodes from the root running the $1,000^{th}$ query 21.5% faster than the AVL Tree (see Figure 5(a)).

Tree	L1	L2	L3	IPC
		Random		
AVL	1108508606	4972838130	252404784	1.094
SPST	267097844	3957313535	135615510	1.385
		Sequential		
AVL	855925856	10890330930	412469096	1.234
SPST	711228747	10479242239	399344564	1.263
		Skewed		
AVL	573854301	3800678199	176536452	1.160
SPST	256760334	3780063118	128213328	1.600

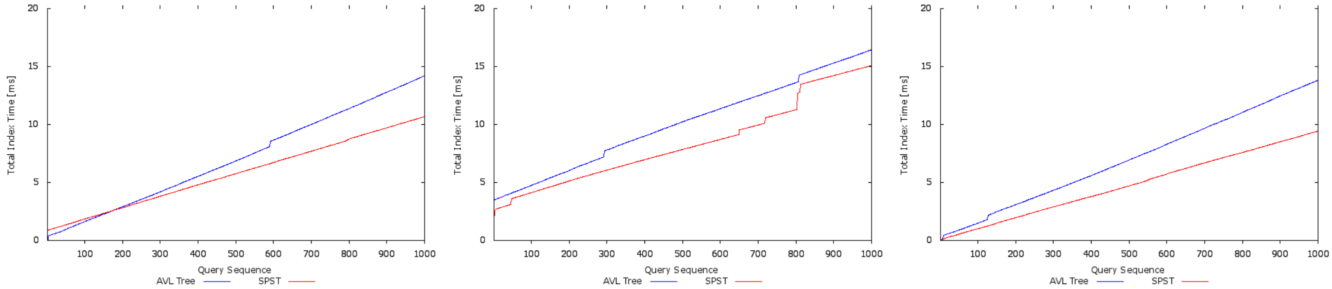
Table 1: Cache Misses and IPC by workload

The sequential pattern was the worst case scenario for the SPST, but still the SPST was 7% faster than the AVL Tree at the $1,000^{th}$ query (see Figure 5(b)). The worst case scenario was the result of many changes in the range predicate of the sequential pattern that required splaying many nodes from the leaves. Over time the SPST mitigated these rotations with 16.9% less cache misses compared to the AVL (see Table 1). The skewed pattern was the best case scenario for the SPST, being 37% faster than the AVL Tree at the $1,000^{th}$ query (see Figure 5(c)). The best case scenario was the result of a skewed workload, achieving an IPC 37% higher. (see Table 1).

4.2 Update Operator

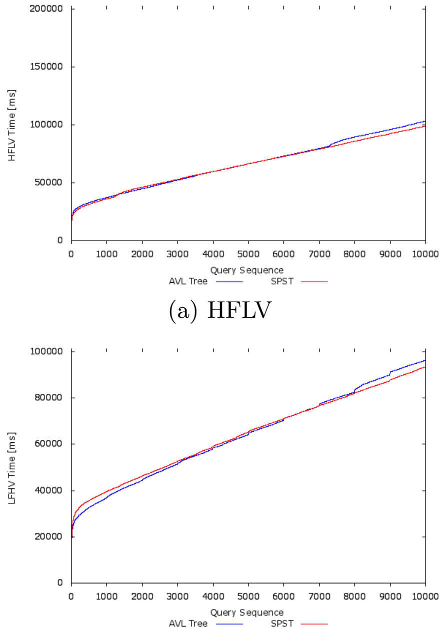
Figures 6(a) and 6(b) depicts the accumulated cracking and update time for the query stream of $10,000$ queries in the HFLV and LFHV scenarios respectively. In both, the SPST achieves the lowest run time. Every time the tree is pruned, updates are boosted but cracking becomes more expensive since we have less nodes to update, but bigger pieces of the cracker column to scan. The SPST was able to prune at convenient moments minimizing the extra cracking cost and greatly boosting update time. For HFLV, we defined empirically a standard deviation of 0.2 milliseconds and for LFHV 200 milliseconds. These values differ because for HFLV it is only analyzed the standard deviation for 10 queries previous to a batch update, while for LFHV 1,000 queries are analyzed. For HFLV, the SPST was pruned only

[1] The cracker index simulator, written in C/C++ and compiled with G++ v.4.7, is available at: www.infosys.uni-saarland.de/research/publications.php



(a) Random Workload (b) Sequential Workload (c) Skewed Workload

Figure 5: Sum of Lookup for Querying and Indexing, and Insertions Time in Various Workloads



(a) HFLV

(b) LFHV

Figure 6: Total time for cracking and updates

once, and was 4% faster than the AVL Tree. For LFHV, the SPST was 5% faster, pruning the tree 8 times and having around 25% of the total size of a full AVL Tree. We observed more rotations in the SPST than in the AVL tree. However, the rotations in the SPST presented less impact in response time compared to the ones in the AVL Tree. While in the SPST the rotations happened most frequently near the root with less cache misses in L1/2/3 and higher IPC, the AVL Tree spanned many rotations usually close to the leaves of the index to rebalance the tree with many unnecessary tree nodes polluting the cache (see Table 1 cache misses).

5. CONCLUSION

This work presented the SPST as a cracker index for database cracking. We explored the Standard Cracking algorithm for select and mixed workloads with three different synthetic patterns where the SPST outperforms the AVL Tree in all scenarios. The SPST was able to cache the most frequently accessed data near to the root reducing cache misses and achieving a higher IPC than the AVL.

In future work, we will compare the SPST with other main-memory index structure for efficiently executing queries on modern processors, like, the recent proposed ART-Tree and Cache-Sensitive Skip List. Our SPST implementation follows the classic splay tree structure, but there are many rotation and pruning strategies that can be explored to improve response time, like, freezing the top of the SPST to diminish the rotations. Furthermore, we focus the SPST on standard cracking. However, there are other cracking approaches in the literature to be explored in future work, like: Hybrid Cracking[7], Sideways Cracking[5] and Stochastic Cracking[3].

Acknowledgments

This work was partly funded by the CNPq Universal, grant 441944/2014-0.

6. REFERENCES

[1] J. Bell and G. Gupta. An evaluation of self-adjusting binary search tree techniques. *Software: Practice and Experience*, 23(4):369–382, 1993.

[2] Elmasri and Navathe. *Fundamentals of Database Systems*. Pearson, 2007.

[3] F. Halim, S. Idreos, P. Karras, and R. H. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *VLDB*, 5(6):502–513, 2012.

[4] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD*, pages 413–424, 2007.

[5] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. *SIGMOD*, pages 297–308, 2009.

[6] S. Idreos, M. L. Kersten, S. Manegold, et al. Database cracking. In *CIDR*, volume 3, pages 1–8, 2007.

[7] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *VLDB*, 4(9):586–597, 2011.

[8] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The uncracked pieces in database cracking. *VLDB*, 7(2):97–108, 2013.

[9] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

In-Memory Spatial Join: The Data Matters!

Sadegh Nobari[†], Qiang Qu[†], Christian S. Jensen[‡]
{nobari,qiang}@siat.ac.cn, csj@cs.aau.dk
[†]Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences
[‡]Department of Computer Science, Aalborg University

ABSTRACT

A spatial join computes all pairs of spatial objects in two data sets satisfying a distance constraint. An increasing demand in applications ranging from human brain analysis to transportation data analysis motivates studies on designing new in-memory spatial join algorithms. Among recent proposals, the following six algorithms can efficiently perform in-memory spatial joins: *Size Separation Spatial Join* (S3), *Spatial Grid Hash join* (SGrid), *TOUCH*, *Partition Based Spatial-Merge Join* (PBSM), *Plane-Sweep Join* (PS), and *Nested-Loop Join* (NL).

This paper addresses the need for studies of aspects that may influence the performance of spatial join algorithms. In particular, given two datasets, A and B, the following aspects may affect performance: the datasets being real or synthetic data, the distributions with respect to density and location of the datasets, and the order of performing the spatial join ($A \bowtie B$ or $B \bowtie A$). To study the effects on performance of these aspects, we implement the six spatial join algorithms in a single framework and conduct extensive experiments.

The findings show that the data being real or synthetic, the data distribution, and the join order can influence substantially the performance of the algorithms. We present detailed findings that offer insight into different facets of each algorithm and that enable comparison across algorithms and datasets. Furthermore, we provide advice on choosing among the spatial join algorithms based on the empirical evaluation.

1. OBJECTIVES

Joins are important in many applications [3, 12, 13]. Spatial joins find spatially close object pairs in two data sets. Spatial joins are employed in many applications, and their use in-memory is becoming increasingly important [9, 13, 14] for two reasons. First, main memory has grown so large that many datasets fit main memory so that spatial joins can be performed entirely in-memory. Second, in-memory join is an unavoidable component of any join since, regardless of whether disk-based or in-memory joins are used, at least some of the join occurs in-memory. Several spatial join algorithms exist. Among them, the following 6 algorithms can efficiently perform in-memory spatial joins: 1) *Size Separation Spatial Join* (S3) [2], an algorithm based on a hierarchy of equi-width grids of increasing granularity; 2) *Spatial Grid Hash join* (SGrid) [4], a sampling algorithm that speeds up building an R-Tree on one dataset; 3) *TOUCH* [9], a hierarchical data-oriented space partitioning in-memory algorithm; 4) *Partition Based Spatial-Merge Join* (PBSM) [10], a multiple assignment algorithm that assigns each spatial object to all partitions it overlaps with; 5) *Plane-Sweep Join* (PS) [11], an algorithm that sorts the datasets in one dimension and scans the datasets synchronously; and 6) *Nested-Loop Join* (NL) [5] that iterates over both spatial datasets in a nested loop and compares all pairs of objects.

Despite many studies on spatial joins, an investigation is missing that studies the importance of the joining data itself. Given two datasets, our empirical study shows that when joining real and synthetic datasets, the data distribution and the order in which the datasets are joined (*join order*) can impact the performance of the algorithms considered very substantially. Our objectives are thus two-fold. First, we implement all the above algorithms in a unified and efficient in-memory spatial join framework in `C++` for ease of comparison and consistent reporting of results. In our framework, we apply implementation optimization and modularity wherever possible. Second, we aim to study how real versus synthetic data, data distribution, and join order influence spatial join performance, and how the findings suggest dirctions for designing robust, high-performance in-memory spatial join algorithms [1, 6, 7, 8, 15].

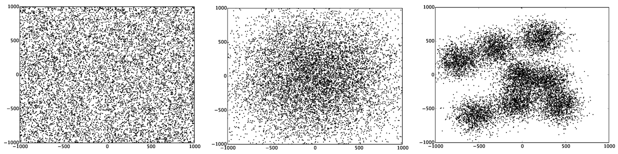
2. ALGORITHMS & IMPLEMENTATION

In the following, we briefly discuss the 6 spatial join algorithms we implemented in-memory. The *Nested Loop* (NL) join [5] compares all pairs of objects from both datasets exhaustively. The *Plane-Sweep* (PS) approach sorts the datasets based on an arbitrary dimension and scans both datasets synchronously. To perform PS, we employ a fast parallel sort to efficiently sort all the objects on one dimension. However, the performance of PS suffers from objects that are not near each other in the other dimensions. Despite its deficiencies, the plane-sweep approach is still broadly used for in-memory joins of the partitions resulting from disk-based spatial joins.

Disk-based approaches first partition both datasets and then join the resulting partitions in-memory via two different approaches, *multiple assignment* and *multiple matching.*
Multiple Assignment: this strategy assigns each spatial

Series ISSN: 2367-2005

10.5441/002/edbt.2017.45



(a) Uniform (b) Gaussian (c) Clustered

Figure 1: Synthetic datasets

object to all partitions it overlaps with. PBSM [10] is an efficient multiple assignment approach that partitions the entire space of both datasets (A & B) into cells using a uniform grid. Then it joins the cells synchronously. SGrid on the other hand, constructs a grid only for the first dataset (A) and probes the intersecting cells in the constructed grid for each object of the other dataset (B).

Multiple Matching: this strategy that assigns each spatial object only to one of the partitions it overlaps with. S3 [2] is a multiple matching approach that maintains a hierarchy of L equi-width grids of increasing granularity. In D dimensions, the grid on a particular level l has $(2^l)^D$ grid cells and assigns each object of both datasets to a grid cell in the lowest level where it only overlaps one cell. Then S3 joins a cell with its counterpart as well as with cells on higher levels for all the cells.

The TOUCH algorithm consists of three steps: Tree construction, assignment, and batch join. TOUCH first builds an R-Tree from the objects of $dataset_A$, i.e., T_A. Then it assigns the objects of $dataset_B$ to the internal nodes of T_A. Finally, the algorithm performs the required pairwise comparisons according to the assignment of the objects of $dataset_B$ to T_A in the batch join step.

3. SETUP AND PRESENTATION

Configuration: The experiments are executed on a Linux Ubuntu 15.04 server equipped with 4 Intel Xeon E5-2650 v3 2.30GHz CPUs and 128GB RAM.

Data description: Our empirical study is done on two *categories* of datasets, namely real (i.e., do not follow any particular distribution) and synthetic. Each synthetic dataset is generated with varying *distributions* of 128,000 cuboids.

The distributions are *Uniform* (Figure 1(a)) (denoted as U), *Gaussian* (Figure 1(b)) (denoted as G), and *Clustered* (Figure 1(c)) (denoted as C) in a universe with a range of [-1000,1000] units per dimension. Figure 1 shows a 2D projection of the distributions. *Uniform* randomly assigns object centers in the bounded universe. *Gaussian* has a mean in the center of universe (0,0,0) and a dispersion of 1000 units. *Clustered* contains 10 clusters. Objects are randomly assigned to one of the cluster centers in the space, and each cluster has a Gaussian distribution ($\sigma = 200$) around the center of the cluster. Each dataset consists only of cuboid objects. After centers of cuboids are distributed, as explained above, each object size is randomly increased uniformly and independently, i.e., the correlation between dimensions is zero. We created 128 thousand objects with average mean length of 10 units.

In addition, to generate the synthetic datasets, we take 10 random samples of 128,000 cylinders (termed R) from Brain datasets (Axons and Dendrites) [9], which do not follow any particular distribution.

To examine the impact of the join order and distribution, we generate (or sample) 20 datasets for each distribution (category). Then we divide the datasets for each distribution into two groups A and B, each with 10 datasets of 128,000 objects. Finally, for each possible combination and order of the datasets, we run all six algorithms and measure the total time. The total time consists of loading time, processing time, and the time for maintaining a result. All the reported total times are averaged over 5 runs under the above-described configuration. In the figures, we distinguish join order (dataset A or B) and the dataset distribution (R, U, C, or G) in the legends. For instance $R_A \bowtie U_B$ means the left hand side dataset is a Real dataset and the right hand side is a Uniform dataset.

Result presentation: We run all 6 algorithms, namely S3, SGrid, TOUCH, PBSM, PS, and NL and report the results in the next section. Since NL was substantially slower than the other algorithms, with a total time of more than 5 hours, with omit NL from the charts. For sake of clarity, we present the results of running the remaining 5 algorithms in the following forms: Since SGrid consistantly performs slower than TOUCH and PBSM and faster than S3 and PS, we present the results for SGrid in Table 1 and the results for the other four algorithms in Figures 1 to 7; In these figures, since TOUCH and PBSM are always faster than S3 and PS, we create two groups (TOUCH and PBSM in one group, S3 and PS on the other group) and use different vertical axes for each group.

4. RESULTS

Our results are summarized as follows:

- The performance of NL and PS is insensitive to the join order.
- When one of the datasets follows a particular distribution or when joining two datasets of the same category, the performance of SGrid is not affected by swapping the datasets. Furthermore, when joining two datasets of different categories, the performance of SGrid varies very substantially when swapping the join order.
- The performance of S3, TOUCH, and PBSM changes when swapping the join order and changing the dataset category of the arguments. As a result, there is not single winner among these algorithms. However, TOUCH does outperform the algorithms in most cases, although in few scenarios, TOUCH can be outperformed by PBSM for some join orders and data distributions. For instance, when joining R and U, the join order of join is critical to the performance of TOUCH. In one order, i.e., $U_B \bowtie R_A$, TOUCH outperforms PBSM while we swap the order, i.e., to $R_A \bowtie U_B$, PBSM outperforms TOUCH.

We proceed to provide detailed observations of each impact separately and finally cover all the impacts together.

4.1 Join order (*order*)

We first observe the impact of swapping the order of the datasets participating in a join, i.e., the *join order*. To observe this impact, we join different samples of the same category of dataset, i.e. real or synthetic, while swapping the join order. The join order has the least impact on PS, while S3 is affected the most. While affected by the join order, TOUCH outperforms all the other algorithms when joining datasets of the same category and distribution.

In Table 1, rows with same dataset, i.e., A=B, tell that SGrid is unaffected by the join order when keeping the dataset category and distribution unchanged. The results of this set of experiments while we join datasets of the same category

Dataset		Total time (s)	
A	B	A ⋈ B	B ⋈ A
Real	Real	142.517	141.37
	Uniform	16.799	48.449
	Gaussian	1.906	2.740
	Clustered	9.662	25.526
Uniform	Uniform	171.782	171.911
	Gaussian	3.579	3.746
	Clustered	41.541	41.196
Gaussian	Gaussian	3.245	3.243
	Clustered	3.450	3.606
Clustered	Clustered	53.706	53.684

Table 1: SGrid performance for varying datasets.

and distribution show that the join order does not affect the relative performance of SGrid. The study thus offers evidence that the join order does not matter when the category and distribution of the datasets are fixed.

4.2 Joining real with synthetic datasets (*category*)

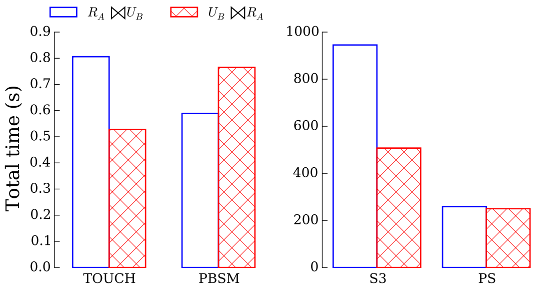
Having observed the impact of changing the join order of datasets of the same category and distribution, we proceed to observe how join order is influenced when we join a real dataset with varying synthetic datasets. Thus, we want to observe how the performance of the join algorithms is affected when one dataset is not following a particular distribution while the other dataset has a particular distribution. Figures 2, 3, and 4 and Table 1 contain results of joining Real (R) dataset with Uniform (U), Clustered (C) and Gaussian (G) datasets, respectively. The performance of all algorithms show a dependence the join order when swapping the category of datasets. The algorithms are affected to the extent that their relative performance changes so that there is no clear winner.

TOUCH and PBSM, in Figures 2 and 4, exhibit an interesting change in their relative performance, so that TOUCH can outperform PBSM only when TOUCH constructs its hierarchical data structure based on the uniform dataset for $U \bowtie R$ and based on the Real dataset for $R \bowtie G$. In all other cases, PBSM outperforms TOUCH. This behavior of TOUCH shows how this algorithm exploits the distribution of the objects. Indeed, TOUCH can outpeform the other algorithms when its constructed tree based on the two datasets is balanced, meaning that similar numbers of objects from both datasets occur in its tree nodes. For instance, when joining Guassian and real datasets, the real dataset can accommodate a balanced distribution of objects of both datasets in the tree constructed by TOUCH. And when joining real and clustered datasets, none of the datasets can yield a balanced tree because the datasets are not mutually aligned in terms of the density of their objects' distribution.

SGrid (Table 1 rows 2, 3, and 4) shows a substantial change, about 3×, in performance when swapping the join order. This indicates that all the algorithms, except the quadratic comparison (NL), are influenced by join order when dataset categories are different. Therefore, it is essential to carefully select the join order when datasets are not following a particular pattern of object distribution, e.g., when joining real datasets.

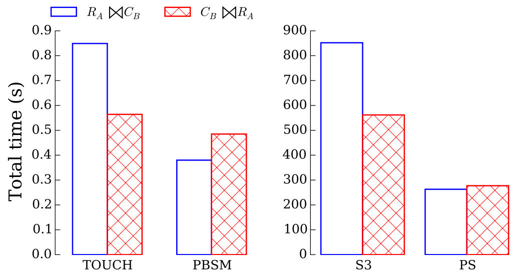
4.3 Joining synthetic datasets (*distribution*)

Finally, we put focus on joining synthetic datasets of different distributions. In other words, we join datasets that follow a particular distribution, namely U, G, and C, while the pattern is different for each. Figures 5, 6, and 7 are results of joining Uniform (U) and Clustered (C) datasets, Uniform (U) and Gaussian (G) datasets, and Clustered (C) and Gaussian (G) datasets, respectively. And rows 6, 7, and 9 in Table 1 show the performance of SGrid for this set of experiments. All algorithms show less impact on the join order when each dataset follows a distribution, in contrast to real datasets with no particular distribution pattern. TOUCH outperforms all the algorithms and it shows a larger change of its performance for different join orders when none of the datasets have a uniform distribution.



(a) R ⋈ U (b) U ⋈ R

Figure 2: Real (R) ⋈ Uniform (U) and vice versa.



(a) R ⋈ C (b) C ⋈ R

Figure 3: Real (R) ⋈ Clustered (C) and vice versa.

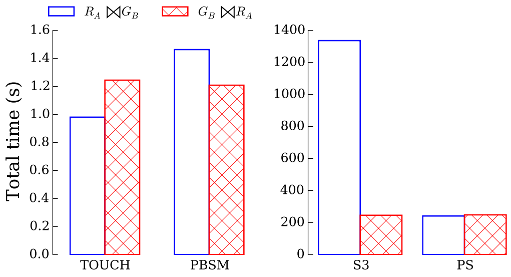
NL is always steadily slower than all the other 5 algorithms, no matter what join order or data distribution we use. For any two of our datasets, NL always requires close to 5 hours to perform the join.

4.4 Overall comparison

The results in Table 1 suggest that when each of the joining datasets follows a particular distribution, the performance of SGrid is not affected when swapping the datasets, i.e., constructing the grid based on the first dataset or the second makes little difference. However, when joining a real dataset with a synthetic dataset, the join order affects the performance of SGrid. This behavior can be explained by the number of cells that an object intersects with as well as the number of objects in the intersecting cells, e.g., the density of the cell.

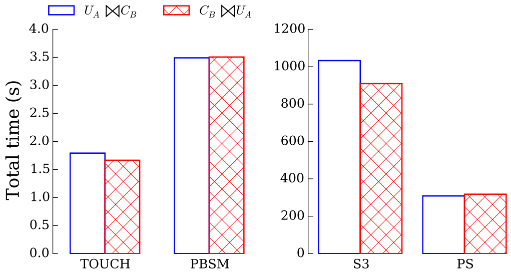
All presented results suggest that the most challenging distribution for all algorithms is that of the Real (R) dataset. The reason is possibly the irregular density/distribution of the objects all over the universe. However, TOUCH and PBSM generally outperform all the other joins due to their ability to handling varying densities.

Among all the algorithms considered, S3 is the most sensitive to the join order, especially when none of the datasets are neither uniform nor real. When both datasets are either uniform or real, the density of objects is consistent with the



(a) R ⋈ G (b) G ⋈ R

Figure 4: Real (R) ⋈ Gaussian (G) and vice versa.



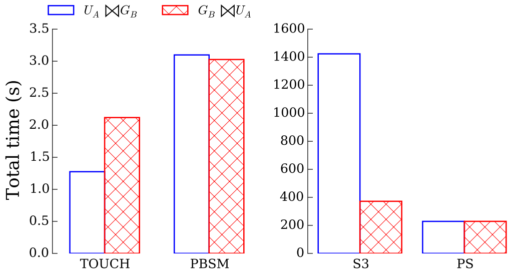
(a) U ⋈ C (b) C ⋈ U

Figure 5: Uniform (U) ⋈ Clustered (C) and vice versa.

region, meaning that similar regions of both datasets have similar object densities. The sensitivity of S3 arises from its dependency on its hierarchical model. Its hierarchy is constructed for the two datasets independently. Therefore, when the order of join changes, the algorithm is affected substantially. In contrast to S3, NL and PS are algorithms that are the least sensitive to the join order. The reasons are that NL uses quadratic comparison to enumerate the search space and that PS sweeps the space one dimension at a time.

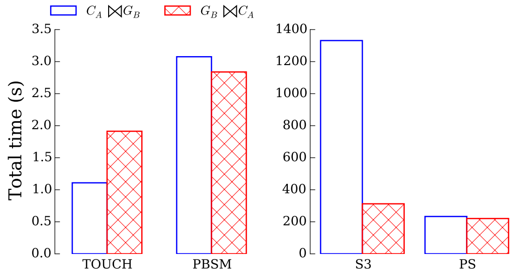
From a practical point of view, TOUCH, SGrid, and PBSM are generally faster than the other algorithms when joining large datasets. SGrid is preferred when the datasets are both real or synthetic with the same distribution. Next, when the datasets are not both real or synthetic with the same distribution, like real and clustered, PBSM is preferred because TOUCH is sensitive to the join order. TOUCH is the fastest when the datasets follow the same distribution. The findings of this paper can help when designing practical algorithms for multi-joins (e.g., $(A \bowtie B) \bowtie C$) and multiple spatial joins (e.g., joining A, B, and C simultaneously). In such algorithms, the order and distribution play even more crucial roles than what we observed in these experiments.

This study suggests that without prior knowledge of the datasets, there is no single winner among the spatial join algorithms. Further, obtaining such knowledge is not always feasible or possible, due to the associated cost or the real-time nature of the datasets, e.g. streaming, intermediate, or growing datasets. Therefore, designing a spatial join algorithm that gradually adapts to the distribution of the argument datasets during the processing and that is capable of changing the order of the joining datasets internally are important for designing robust and scalable spatial join algorithms.



(a) U ⋈ G (b) G ⋈ U

Figure 6: Uniform(U) ⋈ Gaussian(G) and vice versa.



(a) G ⋈ C (b) C ⋈ G

Figure 7: Gaussian(G) ⋈ Clustered(C) and vice versa.

5. REFERENCES

[1] A. Hadian, S. Nobari, B. Minaei-Bidgoli, and Q. Qu. ROLL: fast in-memory generation of gigantic scale-free networks. In *SIGMOD*, pages 1829–1842, 2016.

[2] N. Koudas and K. C. Sevcik. Size Separation Spatial Join. In *SIGMOD '97*.

[3] S. Liu, Q. Qu, and S. Wang. Rationality analytics from trajectories. *TKDD*, 10(1):10:1–10:22, 2015.

[4] M.-L. Lo and C. V. Ravishankar. Spatial Hash-Joins. In *SIGMOD*, pages 247–258, 1996.

[5] P. Mishra and M. H. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

[6] S. Nobari, T. Cao, P. Karras, and S. Bressan. Scalable parallel minimum spanning forest computation. In *PPOPP*, pages 205–214, 2012.

[7] S. Nobari, P. Karras, H. Pang, and S. Bressan. L-opacity: Linkage-aware graph anonymization. In *EDBT*, pages 583–594, 2014.

[8] S. Nobari, X. Lu, P. Karras, and S. Bressan. Fast random graph generation. In *EDBT*, pages 331–342, 2011.

[9] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan, and A. Ailamaki. TOUCH: in-memory spatial join by hierarchical data-oriented partitioning. In *SIGMOD*, pages 701–712, 2013.

[10] J. M. Patel and D. J. DeWitt. Partition Based Spatial-Merge Join. In *SIGMOD*, pages 259–270, 1996.

[11] F. Preparata and M. Shamos. *Computational Geometry: An Introduction*. Springer, 1993.

[12] P. Roy, J. Teubner, and R. Gemulla. Low-latency handshake join. *PVLDB*, 7(9):709–720, 2014.

[13] D. Sidlauskas and C. S. Jensen. Spatial joins in main memory: Implementation matters! *PVLDB*, 8(1):97–100, 2014.

[14] B. Sowell, M. A. V. Salles, T. Cao, A. J. Demers, and J. Gehrke. An experimental analysis of iterated spatial joins in main memory. *PVLDB*, 6(14):1882–1893, 2013.

[15] F. Tauheed, S. Nobari, L. Biveinis, T. Heinis, and A. Ailamaki. Computational neuroscience breakthroughs through innovative data management. In *ADBIS*, pages 14–27, 2013.

Fairness and Transparency in Crowdsourcing

Ria Mae Borromeo, Thomas Laurent,
Motomichi Toyama
Keio University, Yokohama, Japan
{riamae,thomas.laurent,toyama}@keio.jp

Sihem Amer-Yahia
Univ. Grenoble Alpes, CNRS, LIG,
F-38000 Grenoble, France
sihem.amer-yahia@imag.fr

ABSTRACT

Despite the success of crowdsourcing, the question of ethics has not yet been addressed in its entirety. Existing efforts have studied *fairness* in worker compensation and in helping requesters detect malevolent workers. In this paper, we propose fairness axioms that generalize existing work and pave the way to studying fairness for task assignment, task completion, and worker compensation. *Transparency* on the other hand, has been addressed with the development of plug-ins and forums to track workers' performance and rate requesters. Similarly to fairness, we define transparency axioms and advocate the need to address it in a holistic manner by providing declarative specifications. We also discuss how fairness and transparency could be enforced and evaluated in a crowdsourcing platform.

Keywords

Crowdsourcing, Fairness, Declarative Transparency

1. INTRODUCTION

The success of crowdsourcing is undeniable. Many tasks ranging from image recognition to sentiment analysis, are routinely deployed and completed by a pool of workers ready to be solicited. It is therefore timely to start addressing *fairness and transparency* in crowdsourcing, two key questions that are of interest today in ethics.[1] Existing work on fairness has primarily focused on studying worker compensation or on helping requesters identify malevolent workers [2, 17, 21, 20, 19]. For transparency, tools and plug-ins have been developed to disclose computed information such as workers' performance and requesters' ratings [3, 6, 9, 15]. In this paper, we argue that a holistic approach to both fairness and transparency is necessary because of the dependencies between crowdsourcing processes. We define fairness and transparency axioms that serve as a basis for our framework and discuss implementation and evaluation.

Our first endeavor is to understand fair crowdsourcing. Discrimination against individuals is generally defined according to the attributes of those individuals [5]. For example, Google's advertising displays ads for high-income jobs to men much more often than it does to women; and ads for arrest records are most often associated to search queries for common African-American names [18]. In crowdsourcing, even if workers are assigned tasks fairly, the attributes used in task assignment may not have been inferred fairly. It is therefore crucial to characterize fairness in a holistic fashion. We define a set of fairness axioms that capture and generalize existing approaches. For example, we state that in task assignment, two workers with the same qualifications should have access to the same tasks. Similarly, comparable tasks offered by two different requesters should be equally visible to workers. In task completion, fairness to workers means letting them complete tasks without interruption.

Our second question is about transparent crowdsourcing. Intuitively, a crowdsourcing platform that provides better transparency would generate less frustration among workers and see better worker retention. This realization is not new, and several proposals have addressed transparency in crowdsourcing from requester and platform perspectives. Requester transparency reveals details such as recruitment criteria, the conditions under which work may be rejected, and the time before workers' contributions are approved. Platform transparency, e.g., providing feedback to workers on their performance, has also been addressed [12]. Several tools and forums have been developed to disclose information to workers. For example, Turkopticon [9] provides a plug-in to AMT that helps workers determine which HITs do not pay fairly and which requesters have been reviewed by other workers. Turker Nation [2] is an online forum that lets workers exchange information about the latest available HITs and their opinion on requesters. CrowdFlower [3] displays a panel with the worker's estimated accuracy so far. In this paper, we advocate that a single framework is needed to express and enforce transparency. We believe it is essential to provide declarative languages to help requesters and platform developers express what they want to make transparent. Such a solution would also facilitate sharing and comparing transparency choices across platforms.

The question of *how to validate fairness and transparency* in crowdsourcing also merits attention. A common approach is to design appropriate user studies that gather the experience of workers and requesters with specific implementa-

[1] http://www.fatml.org/

[2] http://www.turkernation.com/
[3] https://www.crowdflower.com/

Series ISSN: 2367-2005

10.5441/002/edbt.2017.46

tions of fairness and transparency. Such an approach was used in [12] to validate that feedback contributes to increasing workers' motivation. In this paper, we wish to define a validation protocol based on objective measures and propose to quantify measures such as contributions quality for fairness and worker retention for transparency.

We believe that our proposal paves the way for checking fairness and transparency in existing crowdsourcing systems and also for enforcing them by design in newly developed systems. Section 2 contains a review of fairness and transparency in crowdsourcing and other related areas. Section 3 illustrates the need for fairness and transparency using key use cases, and formalizes our proposal. Section 4 discusses validation.

2. RELATED WORK

2.1 Fairness

Fairness in crowdsourcing has mainly been considered in providing fair wages and managing malicious workers. In [2] and [17], wage discrimination is viewed as the wrongful rejection of work, unfair compensation amount, or delayed payment. In [21], a quality-based reward scheme provides compensation that depends on the quality of a worker's contribution. Vuurens et al. proposed measures to detect and counter malicious users since they observed that nearly 40% of the answers they received from AMT were from malicious users [20].

Studies that address malicious workers through task assignment and worker reputation focus on the quality, reliability, and total cost of worker contributions. Examples of existing task assignment schemes include offering low-cost, reliable answers [7, 11], and accounting for worker skills to maximize the requester's total gain from the completed work [8]. These schemes are requester-centric and do not guarantee fair task assignment to workers.

Overall, we observed that while some work have developed ways of enforcing fair wages and helping requesters detect malicious workers, no holistic approach has been developed to address fairness as a whole for all crowdsourcing processes.

2.2 Transparency

Bederson et al. claimed that higher transparency in working conditions such as hourly wage, or in requester expectations such as work quality metrics, lead to fairness [2]. They asserted the need for requester and platform transparencies to address discrimination but did not tackle its systematic implementation.

Requester transparency has been shown to have positive effects on worker engagement. Studies show that providing workers with information about the requester leads to higher engagement and more effort in task completion [16]. Moreover, providing workers with information about the crowdsourcing workflow and helping them feel part of a group, result in more contributions and higher accuracy [13].

Different initiatives implement transparency in crowdsourcing platforms through plug-ins. Turkopticon [9] is a plugin for AMT that lets workers review tasks and requesters. Crowd-Workers [3] and Turkbench [6] provide expected hourly wages when workers browse tasks. The MobileWorks platform [15] facilitates worker-to-worker communication and assigns manager roles to some workers, allowing workers to monitor each other and benefit from each other's experience, which results in higher quality contributions.

Transparency for workers comes from worker initiatives and communities mainly through forums such as Turker Nation and Mturk Forum[4] where workers share information about tasks, requesters and tools to enhance their experience. These tools are often worker-made scripts that disclose information hidden by the platform such as the time until automatic approval of a submission on AMT.

In summary, workers strive for transparency as it is often fragmented and external to platforms. In this paper, we advocate a systematic way of expressing and enforcing transparency in crowdsourcing platforms through a formalization of fairness and transparency axioms.

3. PROPOSAL

In this section, we first discuss scenarios where discrimination and opacity can hinder workers' and requesters' experience. We then formalize our framework and discuss how we can enforce fairness and transparency in crowdsourcing systems.

3.1 Scenarios

3.1.1 Discrimination

In Task Assignment. Task assignment, the process through which workers find tasks to complete, is central to crowdsourcing. In platforms such as AMT and CrowdFlower, requesters post tasks, and qualified workers choose the ones they like. This simple task assignment mechanism could be characterized as fair because workers have access to the same set of tasks.

Aside from self-appointment, many task assignment algorithms have been designed to optimize a particular objective. However, these algorithms can be discriminatory [14]. For instance, requester-centric task assignment allocates tasks to workers so as to maximize the total gain of the requester. This could be discriminatory to workers. On the other hand, a worker-centric assignment that allocates tasks based on workers' preferences is more likely to be fair to workers, by favoring their expected compensation, but may be unfavorable to requesters.

In Task Completion. In task completion, workers and requesters have different goals. Requesters aim to get enough good results while workers' objectives range from getting paid to improving their skills, spending their time wisely, or signaling their presence and achievements to others [12]. These goals may be advantageous to one but unfair to the other.

For example, in survey tasks, requesters usually publish more HITs than necessary to get a good number of responses. There are cases when a requester cancels tasks when she gets the target number of acceptable responses. Requesters do so to reduce their waiting time and avoid paying more than needed. However, this would be unfair to a worker who has partially completed a task but is not paid for her efforts. A requester may also experience discrimination during task completion in the case of malevolent workers.

[4] http://www.mturkforum.com/

In Worker Compensation. Discriminatory compensation has been identified as one of the major problems for crowd workers [2, 17]. For instance, in AMT, a requester may reject valid work and not pay the worker. In some other cases, a requester promises to provide a bonus when a worker completes a series of tasks but does not do so in the end. In collaborative tasks, a worker may contribute more than another and still receive the same amount of payment.

3.1.2 Opacity

Requester Opacity. In Turker Nation, workers often complain about requesters who reject their contribution without providing feedback. For example, a requester who posts a text summarization task may not publish how a worker's contribution will be evaluated. This requester opacity does not only negatively affect workers' experiences but also affects other crowdsourcing processes. If a contribution is rejected, it is reflected in the worker's history and statistics thus it may limit future task assignment opportunities. If a worker is provided means to post a review of a requester, this may encourage requesters to be more transparent.

Platform Opacity. Since a platform facilitates the entire crowdsourcing process, it must provide valuable information to help requesters and workers achieve their goals [2]. For requesters, it is important to see worker statistics and progress to help them monitor tasks. For workers, it is beneficial to have access to various information that could help them select and complete tasks such as requester reviews and ratings, payment schedules, and estimated worker performance in comparison with other workers. Currently, CrowdFlower shows ratings per task in its task browsing interface and the Turkopticon plug-in shows requester ratings in AMT [9]. Nevertheless, there is currently no systematic way for platform developers and for requesters to specify which information should be made transparent.

3.2 Fairness and Transparency Axioms

In this section, we attempt to define and formalize fairness and transparency axioms. Our proposal does not aim to be exhaustive. Rather, it provides a framework to define and extend a series of axioms that govern checking if a crowdsourcing system abides by fairness and transparency goals, in a principled fashion, or for designing a fair and transparent platform from scratch.

We consider a set of tasks $\mathcal{T} = \{t_1, \ldots, t_n\}$, a set of workers $\mathcal{W} = \{w_1, \ldots, w_p\}$ and a set of skill keywords $\mathcal{S} = \{s_1, \ldots, s_m\}$.

Tasks. A task t is a tuple (id_t, id_r, S_t, d_t) where id_t is a unique task identifier, id_r a unique requester identifier, and S_t is a vector $\langle t(s_1), t(s_2), \ldots, t(s_m)\rangle$ where each $t(s_j)$ is a Boolean value that denotes the requirement or not of having skill s_j to qualify for task t. A reward d_t is given to a worker who completes t. To capture a variety of tasks, skill keywords may be interpreted as expected workers' interests or qualifications.

Workers. A worker w is a tuple (id_w, A_w, C_w, S_w) where id_w is the worker's unique id, A_w is a set of self-declared worker attributes such as demographics and location, C_w is a set of computed worker attributes such as a worker's acceptance ratio, and S_w is a skill vector $\langle w(s_1), \ldots, w(s_m)\rangle$ where each $w(s_j)$ is a Boolean value capturing the interest of w in the skill keyword s_j.

3.2.1 Fairness

We define fairness axioms for task assignment, worker compensation and task completion.

Axiom 1 (Worker fairness in task assignment). *Given two different workers w_i and w_j, if A_{w_i} is similar to A_{w_j} and C_{w_i} is similar to C_{w_j}, and S_{w_i} is similar to S_{w_j}, then w_i and w_j should have access to the same tasks.*

Similarity can be platform-dependent and ranges from perfect equality to threshold-based similarity.

Axiom 2 (Requester fairness in task assignment). *Given two tasks t_i and t_j posted by different requesters id_{r_i} and id_{r_j}, if the required skills for the two tasks S_{t_i} and S_{t_j} are similar, and the two tasks offer comparable rewards d_{t_i} and d_{t_j}, then t_i and t_j should be shown to the same set of workers.*

Skill similarity can be computed using different measures such as cosine similarity.

Axiom 3 (Fairness in worker compensation). *Given two distinct workers w_i and w_j who contributed to the same task t, if their contributions are similar, they should receive the same reward d_t.*

Different measures could be used to compute similarity of contributions depending on the nature of those contributions, e.g., for textual contributions, n-grams could be used [4], for ranked lists, using measures such as Discounted Cumulative Gain [10] would be more appropriate.

Axiom 4 (Requester fairness in task completion). *Requesters must be able to detect workers behaving maliciously during task completion.*

Axiom 5 (Worker fairness in task completion). *A worker who started completing a task should not be interrupted.*

3.2.2 Transparency

It is believed that limiting worker and requester anonymity may lead to fairer labor practices [2]. Therefore, letting workers and requesters reveal information about themselves, ranging from their true identity, to historical worker performance, for example, may help raise everyone's trust in the platform. Transparency axioms govern what requesters and platforms should make available to workers in order to ensure their fair treatment.

Axiom 6 (Requester transparency). *A Requester must make available requester-dependent working conditions such as hourly wage and time between submission of work and payment, and task-dependent working conditions such as recruitment criteria and rejection criteria.*

Axiom 7 (Platform transparency). *The platform must disclose, for each worker w, computed attributes C_w such as performance and acceptance ratio.*

3.3 Implementation

We discuss our preliminary thoughts on how fairness and transparency could be implemented and enforced.

3.3.1 Fairness

Our axioms form a framework to check how fair an existing crowdsourcing system is and also develop guidelines for designing fair crowdsourcing processes from scratch. Using the axioms discussed in Section 3.2, we intend to develop *fairness check* benchmarks and algorithms for existing crowdsourcing systems.

Particular attention needs to be given to checking fairness due to the inter-dependencies between crowdsourcing processes. For instance, an algorithm that checks worker fairness in task assignment must check the fairness of deriving computed attributes such as worker's performance.

3.3.2 Transparency

We advocate the use of a declarative high-level language to specify fairness rules. Such rules can be used by requesters to disclose task requirements, recruitment criteria, evaluation scheme, and payment schedule. Platform designers can use these rules to disclose relevant information that they want to show both workers and requesters. Rules can also be translated into human-readable descriptions for workers' consumption. Last but not least, the declarative nature of those rules will allow easy comparison across platforms. Guiding principles for such a language can be found in works on privacy policy declaration such as in [1].

4. DISCUSSION

4.1 Evaluation

When measuring fairness and transparency, objective measures such as quality of worker contribution and worker retention, can be used in controlled experiments to quantify the level of fairness and transparency of a system as well as its effectiveness.

4.2 Research agenda

Regarding fairness, our immediate agenda is to review existing algorithms for task assignment, strategies for worker compensation, and approaches for task completion, to assess their discriminatory power.

Regarding transparency, we plan to run a user study to validate what kind of transparency choices workers are most sensitive to. Meanwhile, we started designing a declarative language in which transparency rules can be expressed.

5. REFERENCES

[1] M. Y. Becker, A. Malkis, and L. Bussard. A practical generic privacy language. In *ICISS*, pages 125–139. Springer, 2010.

[2] B. B. Bederson and A. J. Quinn. Web workers unite! addressing challenges of online laborers. In *ACM CHI '11 Extended Abstracts on Human Factors in Computing Systems*, New York, NY, USA, 2011.

[3] C. Callison-Burch. Crowd-workers: Aggregating information across turkers to help them find higher paying work. In *Second AAAI*, 2014.

[4] M. Damashek. Gauging similarity with n-grams: Language-independent categorization of text. *Science*, 267(5199):843, 1995.

[5] S. Hajian, F. Bonchi, and C. Castillo. Algorithmic bias: From discrimination discovery to fairness-aware data mining. In *Proceedings of ACM SIGKDD*, pages 2125–2126, 2016.

[6] B. V. Hanrahan, J. K. Willamowski, S. Swaminathan, and D. B. Martin. Turkbench: Rendering the market for turkers. In *Proc. CHI*, pages 1613–1616. ACM, 2015.

[7] C.-J. Ho, S. Jabbari, and J. W. Vaughan. Adaptive task assignment for crowdsourced classification. In *Proc. ICML 2013*, pages 534–542, 2013.

[8] C.-J. Ho and J. W. Vaughan. Online task assignment in crowdsourcing markets. In *AAAI*, volume 12, pages 45–51, 2012.

[9] L. C. Irani and M. Silberman. Turkopticon: interrupting worker invisibility in amazon mechanical turk. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 611–620. ACM, 2013.

[10] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

[11] D. R. Karger, S. Oh, and D. Shah. Budget-optimal task allocation for reliable crowdsourcing systems. *Operations Research*, 62(1):1–24, 2014.

[12] N. Kaufmann, T. Schulze, and D. Veit. More than fun and money. worker motivation in crowdsourcing-a study on mechanical turk. In *AMCIS*, volume 11, pages 1–11, 2011.

[13] P. Kinnaird, L. Dabbish, and S. Kiesler. Workflow transparency in a microtask marketplace. In *Proceedings of the 17th ACM international conference on Supporting group work*, pages 281–284. ACM, 2012.

[14] K. Kirkpatrick. Battling algorithmic bias: how do we ensure algorithms treat us fairly? *Communications of the ACM*, 59(10):16–17, 2016.

[15] A. Kulkarni, P. Gutheim, P. Narula, D. Rolnitzky, T. Parikh, and B. Hartmann. Mobileworks: Designing for quality in a managed crowdsourcing architecture. *IEEE Internet Computing*, 16(5):28–35, 2012.

[16] J. Marlow and L. A. Dabbish. Who's the boss?: requester transparency and motivation in a microtask marketplace. In *CHI'14 Extended Abstracts on Human Factors in Computing Systems*, pages 2533–2538. ACM, 2014.

[17] B. McInnis, D. Cosley, C. Nam, and G. Leshed. Taking a hit: Designing around rejection, mistrust, risk, and workers' experiences in amazon mechanical turk. In *Proc. CHI*, pages 2271–2282. ACM, 2016.

[18] L. Sweeney. Discrimination in online ad delivery. *Commun. ACM*, 56(5):44–54, May 2013.

[19] B. Valeri, S. Elbassuoni, and S. Amer-Yahia. Crowdsourcing reliable ratings for underexposed items. In *Proceedings of WEBIST 2016*, pages 75–86, 2016.

[20] J. Vuurens, A. P. de Vries, and C. Eickhoff. How much spam can you take? an analysis of crowdsourcing results to increase accuracy. In *Proc. ACM SIGIR CIR'11*, pages 21–26, 2011.

[21] J. Wang, P. G. Ipeirotis, and F. Provost. Quality-based pricing for crowdsourced workers. *NYU Working Paper*, 2013.

TASWEET: OPTIMIZING DISJUNCTIVE REGULAR PATH QUERIES IN GRAPH DATABASES

Zahid Abul-Basher
University of Toronto
Toronto, Canada
zahid@mie.utoronto.ca

Nikolay Yakovets
Eindhoven University of Technology
Eindhoven, The Netherlands
hush@tue.nl

Parke Godfrey
York University
Toronto, Canada
godfrey@cse.yorku.ca

Shadi Ghajar-Khosravi
Defence Research and Development Canada
Toronto, Canada
shadi.ghajar@drdc-rddc.gc.ca

Mark H. Chignell
University of Toronto
Toronto, Canada
chignell@mie.utoronto.ca

ABSTRACT

Regular path queries (RPQs) have quickly become a staple to explore graph databases. SPARQL 1.1 includes *property paths*, and so now encompasses RPQs as a fragment. Despite the extreme utility of RPQs, it can be exceedingly difficult for even experts to formulate such queries. It is next to impossible for non-experts to formulate such path queries. As such, several visual query systems (VQSs) have been proposed which simplify the task of constructing path queries by directly manipulating visual objects representing the domain elements. The queries generated by VQSs may, however, have many commonalities that can be exploited to optimize globally. We introduce TASWEET, a framework for optimizing "disjunctive" path queries, which detects the commonalities among the queries to find a globally optimized execution plan over the plan spaces of the constituent RPQs. Our results show savings in edge-walks / time-to-completion of 59%.

1. INTRODUCTION

Military intelligence consists of collecting, analyzing, and extracting intelligence on situations, events, and entities (*e.g.*, people, places, and organizations) from different types of data (*e.g.*, text, video, picture, and signals) produced by a variety of sources (*e.g.*, human, electronic, open source, and sensors) [2]. The goal of the intelligence is to label suspected entities, relationships, and patterns of events. With the emergence of social networks and cyber warfare, graph database systems, especially with visual query facilities, have become a popular choice for military intelligence, as evidenced by the success of products such as GOTHAM by *Palentir*[1] and LINKURIOUS by *Neo4j*.[2]

It has become common methodology to search for pairs of nodes such that each pair is connected by a path—a sequence of labeled edges—matching a path expression expressed via a *path query*. For example, an analyst working on a money laundering case might be searching financial transactions data to find individuals who *own* companies that have transferred money to businesses with investments in off-shore companies, *or* individuals who *work* at said companies. Thus, the analyst is looking for potential links from individuals to off-shore companies matching those path specifications, as illustrated in Figure 1.

In path queries, one can specify the path of interest via a *regular expression*, instead of needing to specify the path explicitly. This is known as a *regular path query* (RPQ). Given the usefulness of path queries, SPARQL 1.1 now supports them via *property paths*. While highly expressive, formulating such queries can be exceedingly difficult, even for experts. As such, visual query systems have been proposed (OptiqueVQS is one such system [7]) to formulate queries which are more user-friendly, intuitive, and less expertise-demanding. Queries generated by VQSs may contain similar sub-queries, and so can be further optimized. We address one such optimization problem: optimizing *disjunctive* regular path queries (dRPQs); that is, a disjunction of a set of regular path queries (RPQs). As the answer set of a dRPQ is the *union* of the answer sets of the individual RPQs, one way to evaluate it is to evaluate the RPQs independently. One can often do better, however, by sharing evaluation of commonalities across the RPQs. Thus, the problem is to optimize multiple regular path queries in graph databases.

In a recent work, Yakovets *et al.* [11] demonstrate that there is a rich plan space for RPQs, and that evaluation cost can differ by orders of magnitude from one plan to another. Their WAVEGUIDE framework is able to find the optimal execution plan with respect to cost-based estimation over this plan space for a given RPQ. Of course for us, when evaluating a dRPQ, choosing the "locally" optimal plans for each of the RPQs may not be globally optimal, given commonalities. One might benefit then by choosing alternative plans that take advantage of the commonalities. We introduce TASWEET, a methodology and prototype for this. TASWEET detects the commonalities over the RPQs (of a dRPQ), then uses WAVEGUIDE with knowledge of the commonalities to find an improved global execution plan (by estimated cost) over the plan spaces of the constituent RPQs.

[1] https://www.palantir.com/palantir-gotham/platform/
[2] https://neo4j.com/developer/guide-data-visualization/

Series ISSN: 2367-2005

10.5441/002/edbt.2017.47

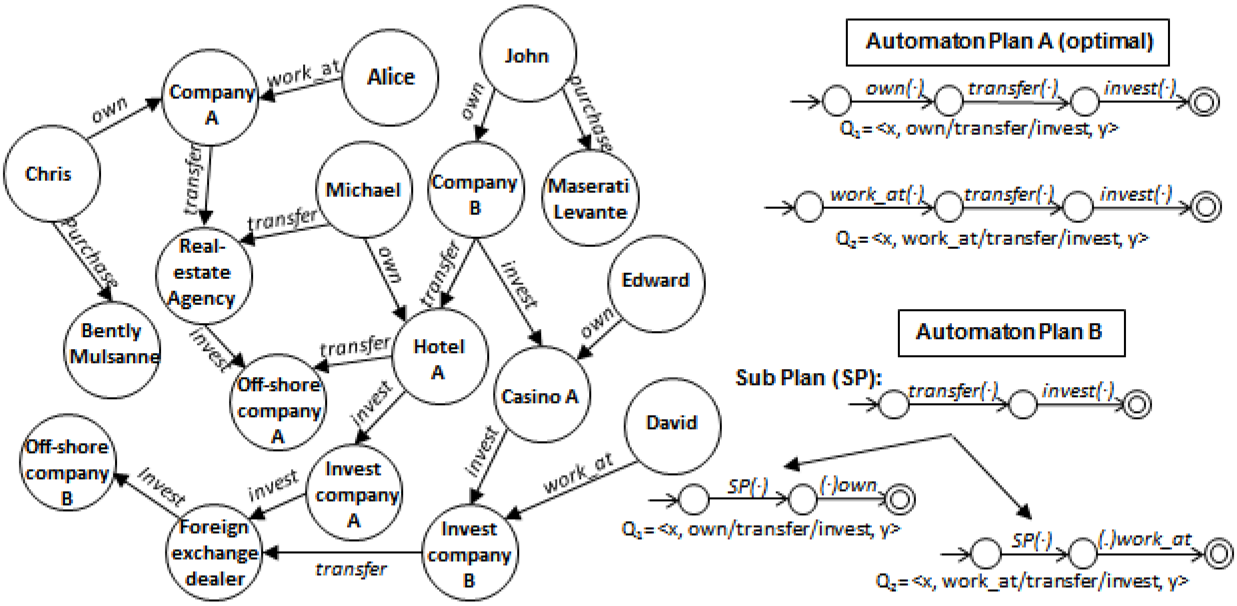


Figure 1: A graph example in military intelligence with two queries Q_1 and Q_2 and their two automaton plans.

2. METHODOLOGY

A regular path query (RPQ) [4] over a graph G is a triple $\langle x, reg, y\rangle$ where x and y are free variables over the nodes N of G, and *reg* is a regular expression. An answer of an RPQ is a node-pair $\langle s, t\rangle$ (with $s, t \in N$) such that there is a sequence of edge labels $a_0 a_1 \cdots a_k$ in G, called a path p, between s and t, and the path p matches the given *reg*. Thus, the *answer set* of an RPQ contains all its answers. The RDF data model and SPARQL query language represent these concepts. With the introduction of *property paths* in SPARQL 1.1, the query language contains RPQs. Herein, we use SPARQL syntax in explanations.

2.1 Evaluation of RPQs

An initial approach to evaluating RPQs was introduced in G+ [4]. There, a *product* finite automaton (FA) is constructed that can be used to navigate and match paths to the regular expression simultaneously. (This approach is nicknamed the FA approach in [11].) More recently, there has been renewed interest in how to evaluate RPQs efficiently in SPARQL over RDF stores [1]. Much focus has been on expanding the relational algebra to leverage relational query optimization and evaluation. It suffices to add an additional operator for transitive closure (called "α"). (This approach is nicknamed as α-RA in [11].) Virtuoso is a relational system that has been extended in this way to support SPARQL and RDF.

The WAVEGUIDE approach generalizes over both the FA and α-RA approaches to provide a very rich plan space for RPQs [11]. While the plan space is exponential (with respect to the length of the regular expression), WAVEGUIDE offers a cost-based optimization via polynomial-time enumeration. In WAVEGUIDE, path search is conducted *simultaneously* while recognizing the path expressions. Its input is a graph database G and a *waveplan* (WP) P_Q which *guides* a number of search *wavefronts* that explore the graph. A *wavefront* is a part of the plan that evaluates breadth-first during the evaluation. Here, the graph exploration is driven by an iterative search strategy which is inspired by the semi-naïve bottom-up procedure used in the evaluation of linear recursive expressions that is based on a *fixpoint*. The key concept is to expand the search wavefronts continuously until there are no new answers; i.e., we reach a fixpoint. Any search wavefront is guided by an *automaton* in the plan, based on a finite state machine FA.

Consider query plans for Q_1, $\langle x, own/transfer/invest, y\rangle$, as in Figure 1. *Plan A* captures the notion of "pipelining". It employs a WP corresponding to a single FA that directly encodes a recognizer for the query's regular expression. Whereas Plan B captures the notion of "materialization". It employs a WP that consists of two subplan automata: the first subplan (SP) is used as a view for transition over states in the second plan. Note that in the second subplan automaton, we used a prepend transition $((\cdot)own)$ over the previous state. The cost for Plan A is 9 edge-walks whereas for Plan B it is 11 edge-walks. In addition to these two plans, there is also a reverse "pipelining" plan (not shown in Figure 1) which is evaluated "backwards", retrieving all pairs connected by edge label *invest* (across such labeled edges in reverse), then prepending with those connected by *transfer*, and finally by *own*, with the plan cost of 12.

The WAVEGUIDE prototype implements this guided graph search as procedural SQL on top of PostgreSQL. However, any type of backend physical graph database model (*e.g.*, triple store or adjacency lists) could be used instead.

2.2 Evaluation of Disjunctive RPQs

Instead of evaluating each RPQ individually, one may benefit by sharing the results of common sub-expressions. Consider the two queries as in Figure 1: Q_1, $\langle x, own/transfer/invest, y\rangle$; and Q_2, $\langle x, work\_at/transfer/invest, y\rangle$. The (locally) optimal automaton plans ("Plan A") do not share any common subplan for Q_1 and Q_2. If we chose sub-optimal automaton plans ("Plan B"), however, then we can share a common sub-plan automaton (*transfer*/*invest*) between the evaluations of Q_1 and Q_2.

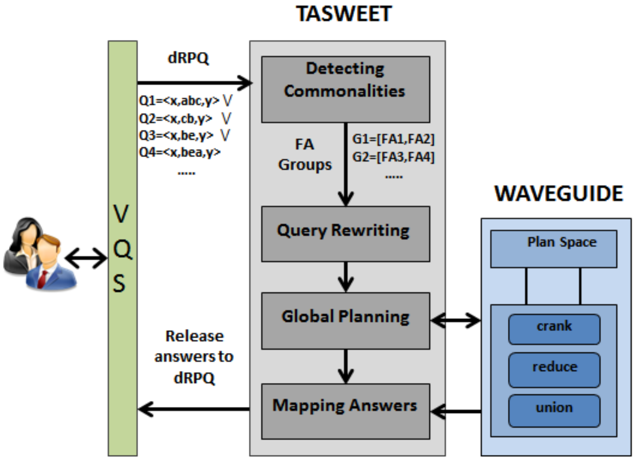


Figure 2: Tasweet Framework

3. THE FRAMEWORK

Our Tasweet framework, shown in Figure 2, takes a dRPQ $\mathcal{D} = Q_1 \vee ... \vee Q_n$ that consists of n RPQs as input over a graph G. Its evaluation is a two-step process: identifying common sub-expressions among queries; and then searching for a global optimal plan such that its cost is less than the total cost of the locally optimal plans of the corresponding RPQs. Figure 2 shows the overall Tasweet architecture.

3.1 Detecting Commonalities

The problem of finding common sub-automata resembles finding the maximum common subgraphs in graphs. The problem is more difficult here, however, as we need to find the largest common subgraphs (sub-automata) for multiple graphs (FAs). We begin our discussion by converting the FA equivalence problem into a graph isomorphism problem. We then extend the concept to find the equivalent sub-automata in FAs using sub-graph isomorphism techniques.

3.1.1 Finite Automata Isomorphism

Deterministic Finite Automata (DFA). A deterministic finite automaton M is a *5-tuple* language acceptor machine, $(S, \sum, \delta, q_0, F)$ where it consists of a set of states (S), a set of input symbols called the alphabet $(\sum)$ and a transition function $(\delta : S \times \sum \rightarrow S)$. There is one state that is marked as an initial state $(q_0 \in S)$ and also there is a set of states marked as accept states $(F \subseteq S)$. Two DFAs are said to be equivalent if they accept the same language.

Minimum Automata. A DFA M is minimal if there is no other DFA N that is equivalent to M with fewer states than M. A DFA M is minimal if (i) all its states can be reached from the initial state q_0 and (ii) no two equivalent states exist. (Two states q_1 and q_2 are equivalent if for all $x \in \sum^*$, $\delta(q_1, x) \in F$ *iff* $\delta(q_2, x) \in F$.) All minimal DFAs for a language L are isomorphic; *i.e.*, they have identical transitions with the same number of states (by the Myhill-Nerode Theorem [5]). By corollary, when joining two minimum DFAs, the structure of the new DFA remains the same. Therefore, we can use the techniques of detecting the maximum common edge subgraph (MCES) to identify the common sub-automata among minimum DFAs; hence, the commonalities among RPQs.

3.1.2 Maximum Common Subautomaton

Most solutions of MCES problem only consider non-labeled edges and nodes in undirected graphs. We adopt the solution from [6] to detect common sub-automata. This has three steps: transforming labeled-graphs into the equivalent line-graphs; producing a product graph from the line-graphs; and detecting the maximal cliques in the product graph, which correspond to MCESs (therefore, common sub-automata).

Constructing Linegraphs. The linegraph $L(G)$ of a graph G is a directed graph where each edge (with label) in original G becomes a node (with the same edge label) in $L(G)$. Two nodes in $L(G)$ are connected with an edge if their corresponding edges in original G share a common node. In the original G, this common node can be an incoming or an outgoing node for two connected edges. That is, it can have four different possibilities depending on the directions of connected edges: source-destination, destination-source, source-source, and destination-destination. Therefore, during our linegraph construction, we store the directions of edges of the common node as edge labels in $L(G)$. Before the construction of linegraphs, we removed all the self-loops in the automaton by creating additional transitions with an additional state. This process is necessary for the next step, clique detection. (Most algorithms for detecting cliques only work with non self-loop graphs.)

Constructing Product Graph. The product graph $L(G_p)$ of two linegraphs, $L(G_1)$ and $L(G_2)$, is constructed as follows [8]. The nodes N_p in $L(G_p)$ are node-pairs defined in the Cartesian product of N_1 of $L(G_1)$ and N_2 of $L(G_2)$. In constructing N_p, we only consider node-pairs that have the same node label of the corresponding linegraphs $L(G_1)$ and $L(G_2)$. The two node-pairs in N_p have an edge in $L(G_p)$ if either *(i)* the same edges exist between the corresponding nodes in the original linegraphs (called a strong connection), or *(ii)* no edges exist between the corresponding nodes (called a weak connection). To reduce the size of the product graph, we remove all nodes with non-common labels among the linegraphs and then build the product graph recursively.

Maximal Cliques in Product Graph. We used the Bron-Kerbosch Algorithm [3] to find all the maximal cliques in the product graph. A maximal clique with a tree that covers strong connections in the product graph corresponds to a maximal common sub-automaton within a group of FAs.

Query Rewriting. A regular expression *reg* can be recognized by several FAs. For example, although the two FAs for the two regular expressions $reg_1 = (xy)^*xz$ and $reg_2 = x(yx)^*z$ are same but the plan space may differ, depending on the chosen FA. In this step, we rewrite FAs based on their shared common sub-automata.

3.2 Global Optimization

In Waveguide, finding the optimal plan for an RPQ is done by dynamic programming, analogous to join enumeration in System R. It works bottom-up to construct a tree which represents the plan. At each level, an optimal plan is selected according to a cost objective; for here, the estimated number of edge-walks. This is calculated based on statistics of the graph (*e.g.*, *n-gram* distributions of the edge labels). In Tasweet, after detecting the common sub-automata, the locally optimal plans for each of them are found by Waveguide. These plans for the common sub-automata then serve as “black-box” views during a second pass with Waveguide to plan each RPQ. The cost of each black-box shared plan is treated as its actual estimated cost divided by the number of queries which share that common sub-automaton. While this is a simplistic, greedy approach, it suffices well for the proof of concept for our methodology. (Immediate future work is to improve this integration of the Tasweet frontend and the Waveguide backend to provide guarantees on the global optimality.)

4. RESULTS & DISCUSSIONS

We provide a preliminary benchmark of our Tasweet framework by considering the optimization of a dRPQ $\mathcal{D}$ which models a typical military intelligence workload: $\mathcal{D}$ disjoins seven realistic, related RPQs—shown in Table 1—over the encyclopedic dataset YAGO2s [9].

Q	
1	?p isMarriedTo/livesIn/isLocatedIn+/dealsWith+ Argentina
2	?p hasChild/livesIn/isLocatedIn+/dealsWith+ Japan
3	?p influences/livesIn/isLocatedIn+/dealsWith+ Sweden
4	?p livesIn/isLocatedIn+/dealsWith+ United_States
5	?p hasSuccessor/livesIn/isLocatedIn+/dealsWith+ India
6	?p hasPredecessor/livesIn/isLocatedIn+/dealsWith+ Germany
7	?p hasAcademicAdvisor/livesIn/isLocatedIn+/dealsWith+ Netherlands

Table 1: Queries used in a military-intelligence dRPQ $\mathcal{D}$.

Query $\mathcal{D}$ finds people-country pairs such that the people related to locations or organizations which have connections with given countries. Tasweet correctly identified a shared sub-plan across all the RPQs in $\mathcal{D}$, "livesIn/isLocatedIn+/dealsWith+". Table 2 shows the difference in the number of edge walks between executing plans in isolation by Waveguide, and as a dRPQ by Tasweet. (Note that fewer edge walks performed result in proportionally faster execution.)

Q	Tasweet	WaveGuide	Δ
1	260	39827	+39373
2	2288	24525	+22178
3	226	33397	+33046
4	4563	269495	+264932
5	2258	40924	+38379
6	4572	42261	+37610
7	4608	7185	+3827
Shared	269895	-	-269895
Total	288670	457614	+169450 (+59%)

Table 2: Deltas in number of edge walks.

None of the (locally) optimal plans for the RPQs of $\mathcal{D}$ evaluate the shared expression (as a sub-plan). Thus, its evaluation is extra work. (This is Shared in Table 2.) But as its materialization can be used by each of the RPQs, it significantly speeds up evaluation of $\mathcal{D}$ overall. Execution of the locally optimal plans cost 59% more edge walks over the execution over the globally optimal plans that materialize and share their common sub-expression. This is excellent proof of concept for our methodology.

For future work, we have clear objectives. First is to develop more robust, efficient means for global optimization. Our second objective is to extend Tasweet to cluster RPQs that arrive together in an application to maximize the commonalities per group. This project is a small step in a grander project to optimize well graph queries. Disjunctive RPQ are a logical step between single RPQs and *conjunctive* RPQs (cRPQs). Thus, our third objective is to extend the Tasweet / Waveguide framework for optimizing cRPQs.

5. CONCLUSIONS

We have presented a proof of concept to show that by exploiting commonalities across RPQs which are meant to be evaluated "in batch" with their answer sets to be unioned—thus, a disjunction of RPQs (a dRPQ)—that their evaluation can be globally optimized significantly beyond locally optimized evaluation of each independently. To do this is challenging. First, the commonalities must be efficiently found. We have developed how to do this, which we overviewed above. Second is how to *push down* reasoning about them into the cost-based optimizer (*e.g.*, Waveguide). Our initial experimentation demonstrated a significant improvement of 59%.

6. REFERENCES

[1] W3c: Resource description framework (rdf). http://www.w3.org/TR/rdf-concepts/,2004.

[2] S. G. Hutchins, P. Pirolli, and S. Card. Use of critical analysis method to conduct a cognitive task analysis of intelligence analysts. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 47, pages 478–482. SAGE Publications, 2003.

[3] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1):1–30, 2001.

[4] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995.

[5] A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.

[6] J. W. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of computer-aided molecular design*, 16(7):521–533, 2002.

[7] A. Soylu, M. Giese, E. Jiménez-Ruiz, E. Kharlamov, D. Zheleznyakov, and I. Horrocks. Optiquevqs: towards an ontology-based visual query system for big data. In *Proceedings of the Fifth International Conference on Management of Emergent Digital EcoSystems*, pages 119–126. ACM, 2013.

[8] P. Vismara and B. Valery. Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. In *Modelling, Computation and Optimization in Information Systems and Management Sciences*, pages 358–368. Springer, 2008.

[9] YAGO2s: A high-quality knowledge base. http://yago-knowledge.org/resource/. Max Planck Institut Informatik.

[10] N. Yakovets, P. Godfrey, and J. Gryz. WAVEGUIDE: evaluating SPARQL property path queries. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, pages 525–528, 2015.

[11] N. Yakovets, P. Godfrey, and J. Gryz. Query planning for evaluating SPARQL property paths. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–15, San Francisco, CA, USA, June 2016.

Poster Paper

A New Division Operator to Handle Complex Objects in Very Large Relational Datasets

André S. Gonzaga
University of São Paulo - ICMC
São Carlos, Brazil
asgonzaga@usp.br

Robson L. F. Cordeiro
University of São Paulo - ICMC
São Carlos, Brazil
robson@icmc.usp.br

ABSTRACT

In Relational Algebra, the operator Division ($\div$) is an intuitive tool used to write queries with the concept of "for all", and thus, it is constantly required in real applications. However, the division does not support many of the needs common to modern applications, particularly those that involve complex data analysis, such as processing images, audio fingerprints, and many other "non-traditional" data types. The main issue is intrinsic comparisons of attribute values, which, by the very definition, are always performed by *identity* ($=$) in the division, while complex data must be compared by *similarity*. Recent works focus on supporting similarity comparison in relational operators, but our work is the first one to treat the division. This paper presents the new **Similarity-aware Division ($\hat{\div}$)** operator. Our novel operator is naturally well suited to answer queries with an idea of "candidate elements and exigencies" to be performed on complex data from real applications of high impact. For example, it can support agriculture, as we demonstrate through a case study in this paper.

CCS Concepts

•Information systems → Query operators; •Theory of computation → Database query processing and optimization; •Applied computing → *Agriculture;*

Keywords

Relational Division; Similarity Comparison; Complex Data

1. INTRODUCTION

The Relational Algebra [3] defines a number of operators to express queries on relations.The Division ($\div$) has an important role in this context because it is the simplest and most intuitive way to represent queries with the idea of "for all", besides being the **only** algebraic operator that directly corresponds to the Universal Quantification ($\forall$) from the Relational Calculus [3]. As a consequence, the division is required in queries that are commonly performed by real applications. For example, it answers the queries as follows: (a) "*What products have* ***all*** *requirements of the industrial quality control?*", (b) "*What bank clients paid* ***all*** *bills of their loans?*", (c) "*What cities have* ***all*** *the requirements to produce a given type of crop?*". Unfortunately, the division is restricted to work with traditional data only.

In this paper, we identify severe limitations on the usability of the division to "non-traditional" data types that are commonly used in modern applications. Today, many real data sets include, besides the traditional numeric values and small texts, more complex data objects such as images, audio, videos, time series, long texts, fingerprints, and many others [8, 6]. One central distinction between traditional and complex data is that the latter must be compared by *similarity*, since comparisons by *identity* are in most cases senseless or unfeasible for data of a more complex nature.

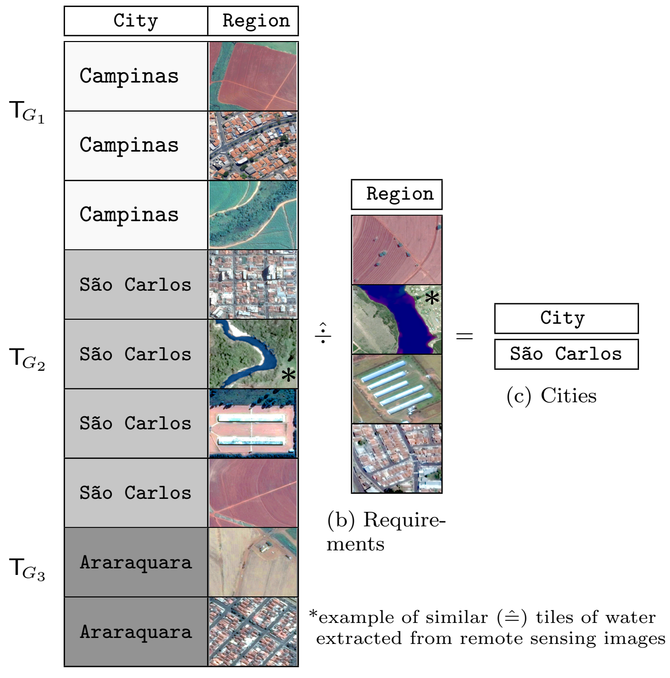
Let us use the query of selecting cities well-suited to produce a given type of crop to exemplify the limitations of the division. Figure 1 illustrates a toy dataset in which the concept of division is required, but the existing operator cannot handle. Relation `CityRegions` describes three cities, i.e., the candidates to produce the crop, each one represented by a set of satellite images taken from regions of the city. For example, the city of Campinas contains regions with bare soil, urban areas, silos and vegetation. Relation `Requirements` describes the needs to produce a given type of crop. In this particular case, we assume that bare soil, water, silos and urban areas are required. The result of dividing `CityRegions` by `Requirements` is relation `Cities`. It contains the list of cities considered appropriate for the crop, that is, those cities that have an image *similar* to each image in `Requirements`. In this particular case, only the city of `São Carlos` satisfies all requirements.

Many researchers have been proposing strategies to support similarity comparison in Relational Database Systems [2], commonly by extending Relational Operators. For example, recent works focus on the Join [5], Selection [4], Grouping [7] and Union [1]. However, to the best of our knowledge, no one focuses on the Division. Here, we tackle the problem by presenting the new **Similarity-aware Division ($\hat{\div}$)** operator. Our main contributions are:

1. **Operator Design and Usability** – we present the first division operator that is well suited to answer queries with an idea of "candidate elements and exigencies" to be performed on complex data from real applications.
2. **Formal Definition and Algorithm** – we formally

Series ISSN: 2367-2005

10.5441/002/edbt.2017.48



(a) CityRegions

(b) Requirements

(c) Cities

*example of similar ($\hat{=}$) tiles of water extracted from remote sensing images

Figure 1: Example of the similarity-aware division used to spot cities suited to produce a crops by analyzing images.

defined the new operator, and carefully designed a fast and scalable algorithm for it;

3. **Experiments** – we validate our proposals by analyzing remote sensing images to support agriculture, following our motivational query with cities and crops. We also performed experiments to show that our algorithm is fast and scalable.

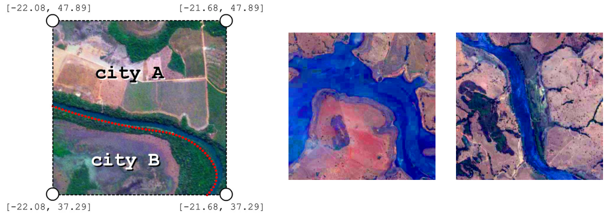
2. RELATIONAL DIVISION

The relational division is expressed by $\mathsf{T}_1\ [L_1 \div L_2]\ \mathsf{T}_2 = \mathsf{T}_R$. Relations, T_1, T_2 and T_R refer to the dividend, the divisor and the quotient, respectively. L_1 and L_2 are lists of attributes from T_1 and T_2, in that order. Both lists must have the same number of attributes, and each attribute in L_1 must be union-compatible with its counterpart in L_2. The quotient relation T_R has all the attributes of T_1 except for those ones listed in L_1. That is, the schema of T_R is given by the relative complement $\overline{L_1}$ of L_1 with respect to $Sch(\mathsf{T}_1)$, i.e., $Sch(\mathsf{T}_R) = \overline{L_1} = Sch(\mathsf{T}_1) - L_1$.

Remember that the quotient in the *arithmetic* operator of division for *integer numbers* is the largest integer that, multiplied by the divisor, defines a value smaller than or equal to the dividend, i.e., $\texttt{quotient} * \texttt{divisor} \leq \texttt{dividend}$. The remainder is the difference between the dividend and the result of multiplying the quotient by the divisor, i.e., $\texttt{dividend} - \texttt{quotient} * \texttt{divisor}$. The relational division is defined in a very similar manner: the quotient relation T_R is the subset of $\pi_{(\overline{L_1})}(\mathsf{T}_1)$ with the largest possible cardinality, such that $\mathsf{T}_R \times \mathsf{T}_2 \subseteq \mathsf{T}_1$. The remainder relation is given by $\mathsf{T}_1 - \mathsf{T}_R \times \mathsf{T}_2$.

3. THE SIMILARITY-AWARE DIVISION

To include comparison by similarity in the Relational Division, we must cover all cases of identity comparisons that are performed intrinsically by the operator. The first case consists in altering the operator to combine the tuples of T_1 with *similar* values in the attributes of $\overline{L_1}$ to form a possible candidate that might be in the result set. The second case lies in relaxing the validation of the requirements by considering as satisfied those requirements in T_2 with values that are similar to their counterparts in T_1. For the first comparison case, consider again the example of selecting cities well-suited to produce a given crop using remote sensing images. One possible approach for this query is shown in Figure 2a. In this case, the images of regions must be grouped by similarity of their latitude and longitude coordinates in order to form candidate cities for evaluation and validation of the given crop requirements. However, a single region could be shared by two different cities. Thus, this tuple containing the image might be assigned to the group of city `A`, of city `B`, of both or, even, of none of them. The second comparison case is shown in Figure 2, where, given the image representing a crop production requirement, its satisfaction should be evaluated through comparison by similarity of the images of regions from the cities.



(a) Example of a region shared by two different cities.

(b) Image representing a region of the city.

(c) Image representing a crop requirement.

Figure 2: Example of comparisons that could be validated in the similarity-aware division shown Figure 1.

3.1 Formal Definition

This section defines formally the new operator involving the concept "for all" based on Relational Division, appending comparisons by similarity between attribute values — called as Division by similarity ($\hat{\div}$).

Definition 1. Two relations T_1 and T_2 are **Union compatible** if and only if they both have the same number of attributes and each attribute from T_1 has the same domain of its counterpart in T_2. We consider A_i to be the i^{th} attribute in the schema $Sch(T)$ of a relation T. The domain of A_i is $Dom(A_i)$. Any two relations T_1 and T_2 are Union compatible if and only if:

$$\begin{aligned} (\,|Sch(T_1)| = |Sch(T_2)|\,) \wedge \\ (\forall A_i \in Sch(T_1), \forall A_j \in Sch(T_2),\ i = j : \\ Dom(A_i) = Dom(A_j)) \end{aligned} \tag{1}$$

Definition 2. The **similarity of attribute values** ($\hat{=}$) is represented as $a_1 \hat{=} a_2$, in which $a_1 \in A_1$ and $a_2 \in A_2$. Attributes A_1 and A_2 must follow the same metric space $M = \langle \mathbb{S}, d \rangle$ with $\mathbb{S}$ being the data domain and $d : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{R}^+$

the distance function, so that $\mathbb{S} = Dom(A_1) = Dom(A_2)$ and $a_1, a_2 \in \mathbb{S}$. For a threshold ξ, values a_1 and a_2 are similar if and only if:

$$a_1 \hat{=} a_2 \Leftrightarrow d(a_1, a_2) \leq \xi \tag{2}$$

Definition 3. The **similarity of tuples ($\triangleq$)** is represented as $t_1 \triangleq t_2$, in which t_1 and t_2 are tuples from relations T_1 and T_2, respectively. One tuple is similar to another if and only if their home relations are Union-compatible, and each attribute of the former has a value that is similar to its counterpart in the latter. We consider $t[A_i]$ to be the value of an attribute A_i for a tuple t. Formally, the similarity of tuples is defined by:

$$\begin{aligned} t_1 \triangleq t_2 \Leftrightarrow \forall A_i \in Sch(T_1), \forall A_j \in Sch(T_2), i = j :& \\ t_1[A_i] \;\hat{=}\; t_2[A_j]& \end{aligned} \tag{3}$$

Definition 4. The **set membership by similarity** ($\hat{\in}$) is represented as $t \hat{\in} T_1$, in which T_1 is a relation and $t \in T$ is a tuple. T_1 and T must be Union-compatible. Tuple t is an element of T_1 by similarity if and only if there exists at least one tuple $t_j \in T_1$ that is similar to t. Formally, we have:

$$t \,\hat{\in}\, T_1 \Leftrightarrow \exists\, t_j \in T_1 : t \triangleq t_j \tag{4}$$

Following the same idea, t is not an element of T_1 by similarity if and only if there is no tuple $t_j \in T_1$ that is similar to t. Formally, it is given by:

$$t \,\hat{\notin}\, T_1 \Leftrightarrow \nexists\, t_j \in T_1 : t \triangleq t_j \tag{5}$$

Definition 5. The **Subset by similarity** ($\hat{\subseteq}$) is represented as $T_1 \;\hat{\subseteq}\; T_2$, in which T_1 and T_2 are Union-compatible relations. Relation T_1 is a subset of T_2 by similarity if and only if every tuple $t_i \in T_1$ is also an element of T_2 by similarity. Formally, its is defined by:

$$T_1 \;\hat{\subseteq}\; T_2 \Leftrightarrow \forall\, t_i \in \mathsf{T}_1 : t_i \,\hat{\in}\, \mathsf{T}_2 \tag{6}$$

Definition 6. The **Difference by similarity** ($\hat{-}$) is a binary operation represented as $T_1 \;\hat{-}\; T_2 \;= T_R$, in which T_1 and T_2 are Union-compatible relations. The resulting relation T_R has all tuples of T_1 that are not members of T_2, by similarity. Formally, we have:

$$T_R = \{t_i : t_i \in T_1 \;\wedge\; t_i \;\hat{\notin}\; T_2\} \tag{7}$$

Definition 7. A **Group of similars** T_{G_k} is a subset of a given relation T_1, such that each of its tuples is similar to at least one other tuple in the group, taking into account only a subset of attributes $L \subseteq Sch(T_1)$. Relation T_{G_k} is also considered to be a group of similars if $|T_{G_k}| = 1$. Formally, T_{G_k} is a group of similars if and only if:

$$\begin{aligned} (T_{G_k} \subseteq\; T_1) \wedge& \\ ((\forall\, t_i \in T_{G_k} \;:\; (\exists\, t_j \in T_{G_k}, i \neq j : t_i[L] \triangleq t_j[L])) \vee& \\ (|T_{G_k}| = 1))& \end{aligned} \tag{8}$$

Definition 8. One **Similarity grouping** T_G is the set of all groups of similars extracted from a relation T_1, taking into account a subset of attributes $L \subseteq Sch(T_1)$. The number of groups is $\kappa = |T_G|$. Formally, we have:

$$T_G = \{T_{G_i} : T_{G_i} \text{ is a group of similars from } \mathsf{T}_1 \text{ regarding } L\} \tag{9}$$

The restriction as follows applies:

$$\mathsf{T}_1 = \bigcup_{k=1}^{\kappa} T_{G_k} \tag{10}$$

Definition 9. The **Similarity-aware division** ($\hat{\div}$) is a binary operation represented as $\mathsf{T}_1\ [L_1 \hat{\div} L_2]\ \mathsf{T}_2 = \mathsf{T}_R$, in which T_1, T_2 and T_R are relations that respectively correspond to the dividend, the divisor and the quotient. $L_1 \subseteq Sch(\mathsf{T}_1)$ and $L_2 \subseteq Sch(\mathsf{T}_2)$ are lists of attributes, so that relations $\pi_{(L_1)}\mathsf{T}_1$ and $\pi_{(L_2)}\mathsf{T}_2$ are Union-compatible. The schema of T_R is defined as $Sch(\mathsf{T}_R) = \overline{L_1} = Sch(\mathsf{T}_1) - L_1$. The instance of T_R is the union of $\pi_{(\overline{L_1})}T_{G_k}$ for the largest possible number of groups of similars $T_{G_k} \in T_G$, such that $\mathsf{T}_R \times \mathsf{T}_2 \;\hat{\subseteq}\; \mathsf{T}_1$. Formally, the quotient T_R is defined as:

$$\mathsf{T}_R = \bigcup_{k=1}^{\kappa} \begin{cases} \pi_{(\overline{L_1})}T_{G_k}, & \textit{if } \forall t_j \in \pi_{(L_2)}(T_2) \;: \\ & (\exists\, t_i \in \pi_{(L_1)}(T_{G_k}) : t_i \triangleq t_j) \\ \varnothing, & \text{otherwise.} \end{cases} \tag{11}$$

The remainder of the similarity-aware division is given by:

$$\mathsf{T}_1 \;\hat{-}\; \mathsf{T}_R \times \mathsf{T}_2 \tag{12}$$

3.2 Algorithm

This section presents a novel algorithm that we carefully developed for the similarity-aware division. It has five input parameters: T_1, L_1, T_2, L_2 and T_G. Parameters T_1 and T_2 are the dividend and the divisor. L_1 and L_2 respectively identify the attributes of relations T_1 and T_2 to be compared with each other, thus defining how to validate the candidate groups with regard to the requirements. For each pair of attributes for comparison, it must be informed the metric to be used to measure similarity and the similarity threshold ξ. Finally, parameter T_G identifies the group T_{G_k} (or the groups) that each tuple of relation T_1 belongs to.

```
SimilarityDivision($\mathsf{T}_1$, $L_1$, $\mathsf{T}_2$, $L_2$, $T_G$);
Result: IDs of the valid groups.
begin
    // all groups are valid at the begining
    $Gv_{id} = \{1,\ 2, \ldots\ |T_G|\}$;
    foreach tuple $t_j \in \pi_{(L_2)}\mathsf{T}_2$ do
        $\mathsf{T} = IndexRangeQuery(\mathsf{T}_1, L_1, t_j)$;
        $Gq_{id} = \varnothing$;
        foreach tuple $t_i \in \mathsf{T}$ do
            $Gq_{id} = Gq_{id}\ \cup\ t_i.groupIDs$;
        end
        $Gv_{id} = Gv_{id}\ \cap\ Gq_{id}$;
    end
    return $Gv_{id}$;
end
```

Algorithm 1: Similarity-aware division.

Algorithm 1 is the pseudocode of the algorithm that we propose. We assume that relation T_1 has indexes known as Metric Access Methods (MAM) ready to be used for the attributes in L_1. The algorithm works by iteratively updating a set Gv_{id} with valid groups' identifiers. All candidate groups are considered to be valid at the beginning, starting with $Gv_{id} = \{1,\ 2, \ldots\ |T_G|\}$. Then, for each requirement t_j from T_2, we perform a range query in T_1 using the requirement itself as the query center and taking advantage of the existing indexes to speed-up the execution. The query finds every tuple $t_i \in \mathsf{T}_1$ such that $t_i[L_1] \mathrel{\hat{=}} t_j$. The next step is to build a set Gq_{id} with all identifiers of the groups that satisfy requirement t_j, that is, the groups of the tuples returned by the query. Then, set Gv_{id} is updated with the intersection of the valid groups from the previous iteration and the groups that meet the current requirement. At the end of the execution, only identifiers of candidate groups that meet all the requirements remain in Gv_{id}.

Time complexity: Algorithm 1 performs $|\mathsf{T}_2|$ range queries in relation T_1. State of the art MAM indexes allow us to perform each range query in $O(log\ |\mathsf{T}_1|)$ time. Therefore, the total runtime of Algorithm 1 is $O(|\mathsf{T}_2|\ log\ |\mathsf{T}_1|)$.

4. EXPERIMENTS

We performed a case study with remote sensing images to allow a semi-automatic identification of cities well-suited to produce particular types of crops. The scheme adopted to perform this study was the same of our example from Figure 1. The remote sensing images come from collaborators of the Centre of Meteorological and Climate Research Applied to Agriculture (CEPAGRI), Brazil, and the Brazilian Agricultural Research Corporation (EMBRAPA). The remote sensing images were already grouped by city limits. The pre-processing started with the segmentation of each original city image, and then we divided it into rectangular region tiles. The dataset contains imagery from five Brazilian cities, i.e., Sapezal-MT, Sorriso-MT, Anhanguera-GO, Catolândia-BA and Volta Redonda-RJ.

For the image comparison between the crop requirements and the region tiles, we use the Earth mover's distance with a similarity threshold of 0.05. The requirements were set as in Figure 3, where four needs for a crop production were given, i.e., (a) silo infrastructure, (b) bare soils, (c) water and (d) urban area to support the production. The recognition of these patterns, was performed through the segmentation of the remote sensing images into six features, i.e., urban area, water, dense vegetation, sparse vegetation, bare soil and silo. After runing the query, only one city had successfully satisfied all these requirements, the city of Sorriso-MT. The other cities were able to satisfy some of the needs, but just this city accomplished all the four requirements. The result of the division algorithm can be validated visually through Figure 4, where we illustrate the city of Sorriso-MT with all requirements and one of the other cities that miss at least one requirement. We also report that our algorithm scaled linearly or even sub-linearly in experiments performed with synthetic data, varying the sizes of the input and output relations up to millions of tuples. Space limitations prevent us from detailing these results.

5. CONCLUSION

In this paper we identified severe limitations on the usability of the Relational Division to process complex data, and tackled the problem by extending it into the new Similarity-aware Division ($\hat{\div}$) operator. We formally defined the new operator and designed a fast and scalable algorithm for it. To validate our proposals, we performed a case study on the support of agriculture. Provided that our algorithm is fast and scalable, we argue that the new similarity-aware division is potentially useful to analyze very large amounts of complex data, even in real-time. For example, it is potentially useful to support agriculture, as presented in this paper; to support genetic analyses, selecting animals that satisfies all the genetic conditions required, and even to help hiring personnel and identifying new clients in enterprises.

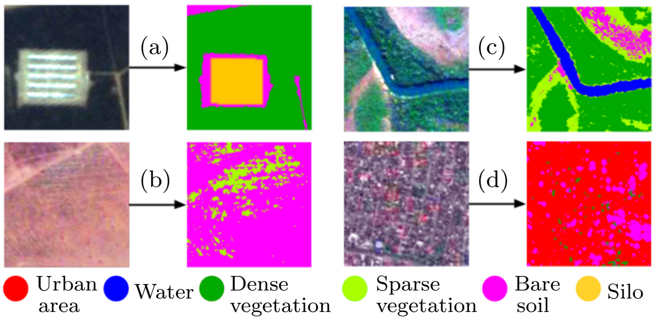
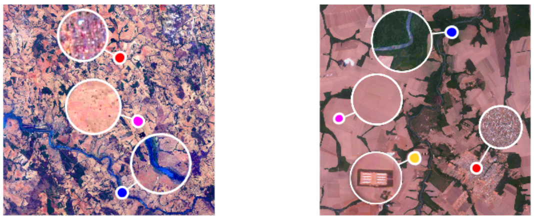


Figure 3: The images selected for the divisor in the case study, representing the requirements for a crop production.



(a) City of Anhanguera-MT missing the silo requirement.

(b) City of Sorriso-MT with all the requirements.

Figure 4: Example of the result over two cities.

6. REFERENCES

[1] W. J. Al Marri, Q. Malluhi, M. Ouzzani, M. Tang, and W. G. Aref. The similarity-aware relational database set operators. *Inf. Syst.*, 59(C):79–93, July 2016.

[2] P. Budíková, M. Batko, and P. Zezula. Query language for complex similarity queries. In *ADBIS*, 85–98, 2012.

[3] E. F. Codd. *The Relational Model for Database Management: Version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[4] Y. Silva, W. Aref, P.-A. Larson, S. Pearson, and M. Ali. Similarity queries: their conceptual evaluation, transformations, and processing. *The VLDB Journal*, 22(3):395–420, 2013.

[5] Y. Silva, S. Pearson, J. Chon, and R. Roberts. Similarity joins: Their implementation and interactions with other database operators. *Inf. Syst.*, 149-162, 2015.

[6] Y.Silva, A. Aly, W. G. Aref, and P.-A. Larson. Simdb: a similarity-aware database system. In *SIGMOD Conference*, 1243–1246, 2010.

[7] M. Tang, R. Y. Tahboub, W. G. Aref, M. J. Atallah, Q. M. Malluhi, M. Ouzzani, and Y. N. Silva. Similarity group-by operators for multi-dimensional relational data. *IEEE TKDE*, 28(2):510–523, 2016.

[8] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search - The Metric Space Approach*, volume 32 of *Advances in DBMS*. Kluwer, 2006.

Protecting Location Privacy in Spatial Crowdsourcing using Encrypted Data

Bozhong Liu
Centre for AI, University of Technology, Sydney, Australia
liu.bo.zhong@gmail.com

Ling Chen
Centre for AI, University of Technology, Sydney, Australia

Xingquan Zhu
Department of Computer & Electrical Engineering and Computer Science Florida Atlantic University, USA

Ying Zhang
Centre for AI, University of Technology, Sydney, Australia

Chengqi Zhang
Centre for AI, University of Technology, Sydney, Australia

Weidong Qiu
School of Information Security and Engineering
Shanghai Jiao Tong University, Shanghai, China

ABSTRACT

In spatial crowdsourcing, spatial tasks are outsourced to a set of workers in proximity of the task locations for efficient assignment. It usually requires workers to disclose their locations, which inevitably raises security concerns about the privacy of the workers' locations. In this paper, we propose a secure SC framework based on encryption, which ensures that workers' location information is never released to any party, yet the system can still assign tasks to workers situated in proximity of each task's location. We solve the challenge of assigning tasks based on encrypted data using *homomorphic encryption*. Moreover, to overcome the efficiency issue, we propose a novel secure indexing technique with a newly devised *SKD-tree* to index encrypted worker locations. Experiments on real-world data evaluate various aspects of the performance of the proposed SC platform.

Keywords

location privacy; spatial crowdsourcing; data encryption; spatial index

1. INTRODUCTION

With the pervasiveness of mobile devices, the ubiquity of wireless network and the improvement of sensing technology, a new mode of crowdsourcing, namely *Spatial Crowdsourcing (SC)*, has emerged [3]. In SC, *task requesters* register through a centralized *spatial crowdsourcing server* (*SC-server*) and request resources related to tasks situated in specific locations. The SC server assigns tasks to registered *workers* according to performance criteria. If a worker accepts the assigned task, he physically travels to the location to perform the required task. Existing SC systems usually require workers to disclose their location in the form of either spatial points or approximate regions, which may have serious privacy implications. For example, with the leakage of location information, an adversary may invoke a broad spectrum of attacks such as physical stalking, identity theft, and breach of sensitive information [7]. Location privacy is therefore a critical security issue and it is important to develop secure SC frameworks to ensure maximum security.

Several approaches have been proposed to protect workers' locations using a trusted third party (TTP) [10]. However, once the TTP is compromised by adversaries, location privacy is infringed. Alternatively, a TTP-free privacy-preserving framework can be achieved by obfuscating each worker's location as a probabilistic distribution, as opposed to a deterministic value [7]. Unfortunately, by simply observing location distributions, the SC server is able to approximately guess a worker's location. In addition, the server knows the final task assignment results, from which it can infer worker locations with reasonable confidence.

The above observations motivate our work, which aims to deliver a general trustworthy SC framework with improved security by requiring workers and requesters to encrypt their location data when registering with and exchanging information through the SC server. Using the correct settings and protocols, real location information is hidden in the ciphertexts and is never disclosed to any party. Accordingly, we can deliver a secure SC framework to sufficiently preserve the location privacy of both workers and requesters based on encryption.

Although encryption provides maximum security protection, the challenge is that the SC server has to assign tasks (*e.g.*, compute the distance between tasks and workers) based on encrypted data. We solve the problem of computation on ciphertexts with a *homomorphic encryption* scheme and introduce a dual SC server design. To address the inefficiency of encryption operations, we propose a secure indexing technique with a newly devised *SKD-tree* to index encrypted worker locations for fast searching and pruning. We have named our SC framework HESI, as it combines a homo-

Series ISSN: 2367-2005 10.5441/002/edbt.2017.49

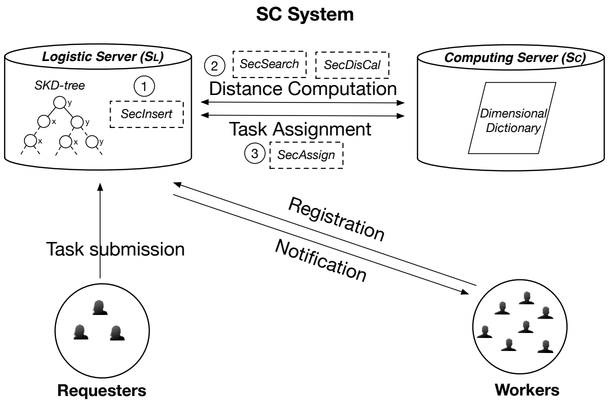


Figure 1: The HESI framework.

morphic encryption (HE) scheme and a secure indexing (SI) technique.

2. THE HESI FRAMEWORK

Our secure SC problem can be considered as a secure outsourcing multi-party computation [5]. Our aim is to enable the SC server to carry out all the computations while users (workers/requesters) do nothing but perform a small number of encryptions and decryptions.

2.1 The Dual-Server Architecture

We have adopted a *dual-server* design and propose an SC framework that consists of two non-colluding semi-honest servers. The assumption of non-collusion between two service providers, such as Google and Amazon, is reasonable in practice [8], because the collusion of two well-established companies may damage their reputation and consequently reduce their revenues. According to the semi-honest model, these two servers are curious but will follow the protocols. A dual-assisted server setting can liberate users from heavy computation and communication by allowing the server to complete computation tasks. The security intuition behind dual-server settings has been addressed in related domains such as secure multi-party computation [9], secure kNN search [2] and secure trajectory computation [4], and we refer interested readers to these texts for further rationale.

Figure 1 illustrates the HESI framework. It consists of workers, requesters, and two servers: the *logistics server* (S_L) and the encrypted data *computing server* (S_C). In general, the requesters submit their tasks to the SC platform and the tasks are dispatched to appropriate workers. The S_L handles all logistics issues, including new user/task registration, data indexing maintenance (*i.e.*, SKD tree), and task assignment, whereas the computing server is an auxiliary server that handles the computation of encrypted data. We assume that each server owns a pair of encryption keys, *i.e.*, (pk_L, sk_L) and (pk_C, sk_C), where pk is the public key and the private key sk is known only to owner.

Let $\mathcal{W}$ and $\mathcal{T}$ denote the set of workers and tasks, respectively. Each worker only accepts a limited number of tasks at the same time and accepts only those within a certain distance. The whole process is presented as follows:

(1) Worker Registration (WR). Workers register and sign up with the S_L. The S_L will index all workers' encrypted locations using an interactive secure indexing (SI) technique and store them in a data structure called an SKD-tree, as shown in Figure 1.

Table 1: The outline of the secure protocols

Scope	Protocols	Description
Index	<u>*SecInsert*</u>	Insert a node (worker's encrypted location) into an SKD-tree securely.
	<u>*SecSearch*</u>	Given a spatial range, output a set of workers within this range.
Computation	<u>*SecDisCal*</u>	Calculate the distance of two locations securely.
Assignment	<u>*SecAssign*</u>	Perform secure task assignment according to some strategies.

(2) Task Submission (TS). Apart from worker locations, our framework also protects the privacy of requester/task locations. The requesters submit their tasks to the S_L, including the task location and task mission.

(3) Distance Computation (DC). In order to assign workers to tasks in close proximity, the SC platform needs to know the distances between tasks and workers. The S_L and S_C together perform an interactive protocol to compute the distance based on the encrypted data using homomorphic encryption scheme [6]. The distance between tasks and workers can be used for evaluation during task assignment, while the real locations of tasks and workers are never revealed to either the S_L or S_C. In addition, neither the S_L and S_C are able to learn any sensitive information from the intermediate result, unless they conspire which is not allowed according to the protocol.

(4) Task Assignment (TA). Based on a distance matrix $\mathcal{M}$, where element m_{ij} represents the distance between task t_i and worker w_j, and the task acceptance conditions of workers, the SC-system assigns the tasks to workers with the goal of maximizing the number of assigned tasks while minimizing the workers' travel costs. The task assignment is carried out by an interactive protocol between the S_L and the S_C, under conditions that both servers cannot learn any sensitive information from the intermediate results.

(5) Task Notification (TN). The final stage is for the S_L to notify assignment results to corresponding workers. Because the S_L does not know the exact locations of the tasks, it needs to communicate with requesters as follows. The S_L first sends the workers' public keys to the requester according to the assignment results. For example, if a result record is $\{t_i : w_1, \ldots, w_k\}$, t_i will receive the corresponding workers' public keys. Next, the requester encrypts the task's location with the received public keys and sends them back to the S_L. The S_L then notifies the worker w_j with a message containing encrypted task locatiosn and task mission. When w_j receives the message, he decrypts the ciphertext using his own private key to obtain the task's location. The worker is then able to travel to the specified location and perform the task according to the mission.

We briefly outline our proposed secure protocols in Table 1. These secure protocols are categorized into distance computation, secure index and assignment. There are four main protocols[1]: *SecDisCal*, *SecInsert*, *SecSearch* and *SecAssign*. In the following, we focus on explaining the details of our proposed secure indexing.

[1] The details of the protocols can be found in a full version of this paper at https://goo.gl/1jkqCn

2.2 Secure Indexing

Given only a small number of workers usually satisfy the neighborhood condition of a task, instead of comparing all worker-task pairs, the encrypted locations of workers are indexed in advance, and unpromising workers are pruned before computing the distances.

KD-tree [1] was the first, and most promising, indexing technique we considered to tackle this purpose. Using a KD-tree to construct our framework presents two major challenges. First, all operations must be performed on encrypted data, to ensure that neither the S_L nor the S_C will obtain any private location information during the indexing process. Second, normal KD-trees hold a potential privacy threat. The splitting dimensions of normal KD-trees are pre-determined and public – nodes in odd levels split the space with the x-dimension, and nodes in even levels split with y-dimension. Consequently, the S_L could deduce the relative locations of all workers. For instance, it knows w_1 is to the left side of w_2 if w_2 is an x-splitting node and w_1 is in the left subtree of w_2. By continually observing the tree, the possible spatial range of w_1 can be shrunk to a small region, if enough relative location information is collected. Even though the location information is relative, not exact, it is still insecure.

We have therefore developed a novel secure indexing technique based on KD-tree, called SKD-tree. One major difference between an SKD-tree and a normal KD-tree is that SKD-tree is split into two parts and stored on the S_L and S_C separately. The S_L stores the tree structure information (*i.e.*, parent-child relationships) while the S_C stores the dimension splitting information in a dictionary. The splitting dimension of each node is selected randomly, allowing nodes in the same level to split the space along different dimensions. This feature improves security by increasing the difficulty of inferring the relative locations.

Figure 2 compares a normal KD-tree with an SKD-tree. All worker locations are indexed by a normal KD-tree (left) and an SKD-tree (right). Each line in the graph represents a node that splits the space along a particular dimension. In a normal KD-tree the splitting information is fixed in advance. By contrast, each node's splitting dimension is randomly generated and separately stored in the SKD-tree on the S_C. The S_L, preserving the tree structure, acquires the dimension splitting information from the S_C through secure protocols. The shaded nodes in the SKD-tree represent encrypted data. For example, while node ① at the level 0 (*i.e.* root) is partitioned along x axis and node ② at the level 1 is partitioned along y axis in the normal KD-tree, both node ① and ② in our SKD-tree are partitioned along x axis according to the dimension dictionary stored at the S_C, which increases the difficulty for malicious adversaries to infer the relative locations of workers.

Our SKD-tree is constructed by inserting nodes one by one using the protocol *SecInsert*. By quering the SKD-tree using *SecSearch*, we can obtain promising neighborhood workers efficiently. In practice, the size of potential worker set will be small as there are only limited number of workers close to a task, which means generally a great number of nodes can be pruned during each iteration.

3. PERFORMANCE EVALUATION

3.1 Benchmark Data

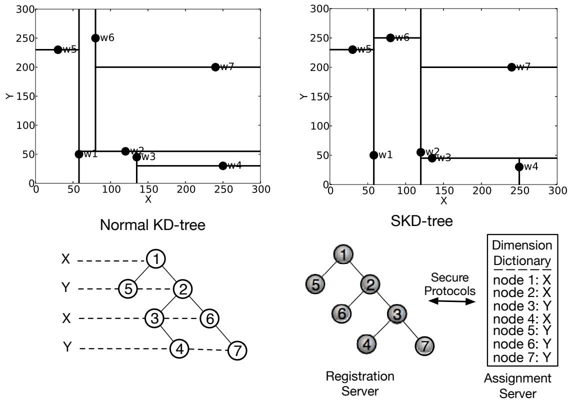


Figure 2: An example of a normal KD-tree *vs.* an SKD-tree.

The **Yelp dataset**[2] is a collection of user reviews about local businesses, such as restaurants. It includes users' comments, check-ins and business information. We consider each Yelp user as an SC worker with their check-in as the location, and assume that the restaurants are the specified task targets. The **Gowalla dataset**[3] is a location-based social network dataset where users share their locations with their friends. Each Gowalla user is considered to be an SC worker, and their location is the most recent check-in. Each check-in point is also modeled as a task location.

In our experiments, $10,000$ workers and $5,000$ tasks were chosen from both datasets. It is assumed that each worker's maximum number of tasks (T_i) and maximum travel distance (D_i) are the same. By default, we set T_i to 5, set $D_i = 1km$ for the Yelp dataset and $D_i = 10km$ for the Gowalla dataset. We ran each experiment five times and report the average runtime.

3.2 Experimental Results

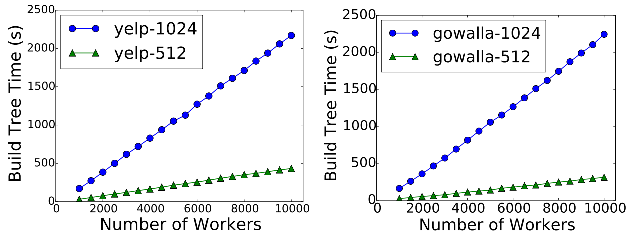
3.2.1 SKD-tree Evaluation

We first evaluate the scalability of our tree construction method. Two Paillier key sizes are used: $K = 512$ bits and $K = 1024$ bits. We vary the number of workers from 1000 to $10,000$ and record the corresponding runtime for building the SKD-tree. The results are presented in Figure 3 and show that the runtime for tree construction achieves good scalability on both datasets. In addition, we observe that the encryption key size influences the performance significantly, which justifies the fact that a trade-off between privacy and efficiency exists. In the following experiments, we used 1024 as the default key size.

Next we evaluate the operations on SKD-tree. Because deletion is very similar to insertion, only the results of insertion and range search are presented. Figure 4a illustrates the costs of inserting a node into trees of different size. The y-axis represents the average runtime of inserting a node (*i.e.*, worker) into the tree. It can be observed that the trend increases quite slowly, showing that the insertion operation is scalable to the tree size.

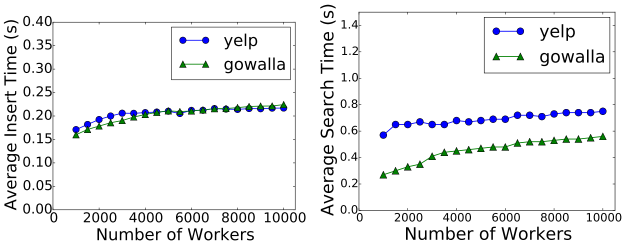
[2] https://www.yelp.com/dataset_challenge

[3] https://snap.stanford.edu/data/loc-gowalla.html



(a) Build Tree - Yelp (b) Build Tree - Gowalla

Figure 3: Evaluation of tree construction



(a) Insert Time vs. Tree Size (b) Search Time vs. Tree Size

Figure 4: Tree operation evaluation

To evaluate range search, we invoked 100 random queries with a default range size of $1km$. Figure 4b reports the average search time with respect to the number of workers, which demonstrates good scalability on both datasets. Moreover, it costs less time to search the Gowalla dataset because it is sparser than the Yelp dataset, more unnecessary comparisons are pruned at each step.

3.2.2 Overall Performance Evaluation

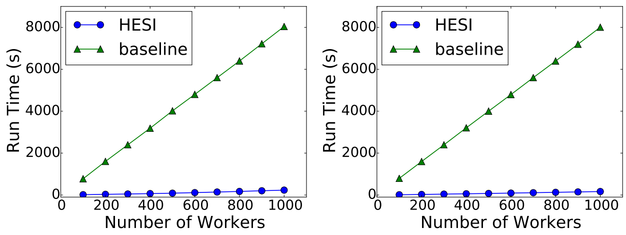
We compare the performance of HESI to the baseline index-free framework which compares all worker-task pairs for distance information. We set the number of tasks as 10 and vary the number of workers from 100 to 1000. It can be seen in Figure 5 that the performance of the baseline framework increases linearly with respect to the number of workers. The HESI framework shows a clear advantage over the baseline. This is mainly because HESI is able to prune a large number of unnecessary distance computations with the contribution of the secure indexing technique.

3.2.3 Communication Cost Evaluation

We evaluate the communication overhead between two servers in the proposed framework. Specifically, we record the total size of data that transferred during the executions of the secure protocols. Table 2 shows the results with respect to different numbers of workers and tasks. It can be seen that the cost changes from 8.18MB to 26.25MB when the number of workers varies from 2000 to 10000, and changes from 5.24MB to 21.83MB when the number of tasks varies from 1000 to 5000. The result shows that the extra communication overhead due to the dual-server design is acceptable, and our proposed framework is feasible in practice.

4. CONCLUSIONS

In this paper, we proposed a novel privacy-preserving framework for spatial crowdsourcing, which ensures that user locations are never released to anyone, yet the system is still able to assign tasks to workers in an efficient way. The key innovation of our framework, compared to existing work in the field, is threefold: (1) a new encrypted data-based spatial crowdsourcing framework for the SC community; (2) a secure SKD-tree structure to store and index encrypted data for fast search; and (3) ensured data privacy (including worker and requester privacy) and data security, whereas existing works only limit to worker privacy.



(a) Yelp (b) Gowalla

Figure 5: Performance Improvement

Table 2: Communication cost

#Workers	Cost (MB)	#Tasks	Cost (MB)
2000	8.18	1000	5.24
4000	10.95	2000	8.63
6000	16.20	3000	13.35
8000	20.22	4000	17.39
10000	26.25	5000	21.83

Acknowledgement This work was supported by the Australia Research Council (ARC) Discovery Project under Grant No. DP140100545 and program of New Century Excellent Talents in University under Grant No. NCET-12-0358.

5. REFERENCES

[1] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, 1975.

[2] Y. Elmehdwi, B. K. Samanthula, and W. Jiang. Secure k-nearest Neighbor Query over Encrypted Data in Outsourced Environments. In *ICDE*, pages 664–675, 2014.

[3] L. Kazemi and C. Shahabi. Geocrowd: Enabling query answering with spatial crowdsourcing. In *SIGSPATIAL*, pages 189–198, 2012.

[4] A. Liu, K. Zheng, L. Li, G. Liu, L. Zhao, and X. Zhou. Efficient Secure Similarity Computation on Encrypted Trajectory Data. In *ICDE*, pages 66–77, 2015.

[5] J. Loftus and N. P. Smart. Secure Outsourced Computation. In *AFRICACRYPT*, pages 1–20, 2011.

[6] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT*, pages 223–238, 1999.

[7] L. Pournajaf, L. Xiong, V. S. Sunderam, and S. Goryczka. Spatial Task Assignment for Crowd Sensing with Cloaked Locations. In *MDM*, pages 73–82, 2014.

[8] B. K. Samanthula, F. Rao, E. Bertino, and X. Yi. Privacy-Preserving Protocols for Shortest Path Discovery over Outsourced Encrypted Graph Data. In *IRI*, pages 427–434, 2015.

[9] Y. Sun, Q. Wen, Y. Zhang, H. Zhang, Z. Jin, and W. Li. Two-Cloud-Servers-Assisted Secure Outsourcing Multiparty Computation. *The Scientific World Journal*, 2014, 2014.

[10] H. To, G. Ghinita, and C. Shahabi. A Framework for Protecting Worker Location Privacy in Spatial Crowdsourcing. *PVLDB*, 7(10):919–930, 2014.

Break the Windows: Explicit State Management for Stream Processing Systems

Alessandro Margara
DEIB, Politecnico di Milano
alessandro.margara@polimi.it

Daniele Dell'Aglio
IFI, University of Zurich
dellaglio@ifi.uzh.ch

Abraham Bernstein
IFI, University of Zurich
bernstein@ifi.uzh.ch

ABSTRACT

Several stream processing and reasoning systems have emerged in the last decade, motivated by the need to process large volumes of data on the fly, as they are generated, to timely extract relevant knowledge. Despite their differences, all these systems isolate the data that is relevant for processing using (fixed size) windows that typically capture the most recent data and assume its validity.

We claim that this paradigm is not flexible enough to effectively model several application domains and we propose a novel abstraction that enables for explicit state representation and management. We model state as a collection of data elements annotated with their time of validity and we augment the traditional stream processing paradigm with state-handling abstractions to declare how the input streams affect the state of the system and how the state influences the results of the processing.

Keywords

Stream Processing; Stream Reasoning; Event Processing; Windows; State Management; Explicit State

1. INTRODUCTION

Several application domains require analyzing *streams* of data on-the-fly, as new data become available, to extract valuable knowledge. Examples include environmental monitoring, click stream analysis in Web sites, traffic monitoring, credit card fraud detection, computer systems monitoring, interaction analysis in social media, and smart cities.

In the last decade, this need led to a bloom of technologies for *stream processing* and *reasoning* [6, 12], which introduce (i) languages and programming abstractions to define how to extract relevant knowledge from the input data; (ii) algorithms and techniques to efficiently perform such task.

Stream processing and reasoning systems were developed by researchers and practitioners active in diverse fields, such as database systems [3], event-based systems [11], knowledge representation [5], and Big Data processing [13]. As a consequence, they present heterogeneous design choices and characteristics.

Despite their differences, all these technologies build on the implicit assumption that the most recent data is also the most relevant for processing, and isolate recent data using *window* operators, for instance to limit the scope of the analysis to the last ten elements of the stream or to the elements that occurred within the last five minutes. Furthermore, they often assume that *all* the information stored in windows is valid when the processing takes place.

We believe that this paradigm is not flexible enough to effectively model several application domains, since: (i) windows with a predefined and fixed size might not be suitable to define the portion of streaming data that is relevant for processing, which might depend on the specific content of the data; (ii) windows might include invalid or contradictory information.

To motivate our claims, let us refer to some concrete use cases. Consider a click-stream monitoring system that analyzes the interactions of potential customers with an e-commerce Web site. The system should trace a user from the moment when she enters the Web site to the moment when she leaves the Web site. A shorter observation time frame would be meaningless for the application logic, whereas a larger time frame could waste computational resources. This simple example highlights that fixed-size windows are not always adequate to isolate relevant stream elements.

Consider now a security service to monitor the position of visitors in a building, in which sensors signal a new event every time a visitor enters a room. If we assume a fixed time-window of five minutes, it is possible that a visitor moves through multiple rooms within the scope of a single window. Considering all the events generated within this fixed time frame as valid would lead to the erroneous conclusion that the visitor is simultaneously in multiple rooms. This example shows that one might infer contradictory information if she simply considers as valid all the data elements within a window, without properly considering their mutual relations —in our example, the most recent position invalidates and updates any previous position of the same visitor—.

We believe that the above limitations could be overcome with flexible abstractions to model *state* in stream processing systems. For instance, the e-commerce scenario could benefit from the presence of state information that records which users are active at a given point in time. Similarly, the security service could model the position of each visitor as part of the state and update such a state whenever the visitors move.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.50

Moving from these premises, we propose an extension to stream processing and reasoning that makes state explicit and captures it as a first class object. We model state as a collection of data elements annotated with their time of validity. Then, we augment the traditional stream processing model that defines transformations from input streams to output streams with state-handling abstractions to (i) declare how state information influences the results of the processing —for instance, we want to monitor only active users in the e-commerce scenario, where the set of active users is defined as part of the state—; (ii) declare how the stream of input data updates the state —for instance, a new event in the security service invalidates previous information about the position of a visitor and adds a new state element with the current position of that visitor—.

We see several benefits from explicit state management in stream processing systems. As motivated by the scenarios above, the possibility to define and reference state information has the potential to ease the modeling of the application at hand. Furthermore, it might simplify the processing task by activating some derivations only when specific conditions on the state are met. Finally, the presence of an explicit repository for state information would make the state available for query and retrieval.

The paper is organized as follows. Section 2 presents the state-of-the-art systems for stream processing and reasoning, with emphasis on their approaches to manage state information. Section 3 presents the model we propose to explicitly manage state. Section 4 surveys work that is related to the topic of this paper, and Section 5 concludes the paper and draws a road map for future work.

2. BACKGROUND

This section surveys the main models and technologies for stream processing and reasoning, focusing on their state management capabilities.

Perhaps the first processing model for streaming data is CQL —Continuous Query Language— that builds on the relational model [3]. CQL introduces *window* operators to isolate finite blocks of the input streams and then applies relational processing on such blocks. Windows typically include the latest elements received from the input streams: as new data is received, the content of the windows changes, and the processing is re-executed to update the results accordingly. The most widely adopted windows have a fixed size in terms of number of elements —count windows— or time span —time windows—. This model is the core of virtually all Data Stream Processing Systems (DSMSs) [4].

The relational core of this model facilitates the interoperability of streaming data and static relational tables. Although static tables could be theoretically employed to store state information, the model does not include state-management functionalities.

Complex Event Processing (CEP) systems consider each stream element as the notification of occurrence of an *event* at some points in time, and offer abstractions to define situations of interest as (temporal) patterns of events [6, 11]. Patterns are typically searched for within time windows or include temporal constraints conceptually similar to time windows.

Some CEP systems adopt interval time semantics, meaning that the situations detected from the raw events can have an associated time *interval* of validity [2]. Situations con encode the current state of the application environment and be composed with further events during processing. Nevertheless, these systems do not offer specific abstractions to model and update state information. Furthermore, situations are not persisted and cannot be queried.

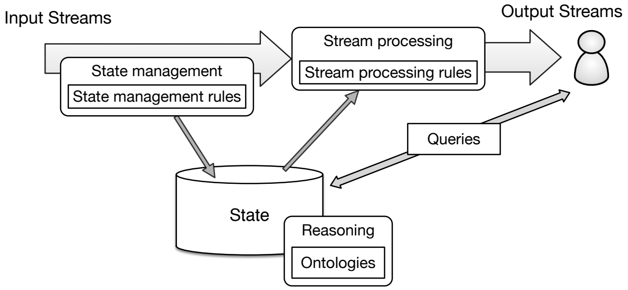


Figure 1: Model of stream processing with explicit state management

Stream reasoning extends the above approaches by exploiting the RDF data model to represent data elements within the streams, which enables for complex forms of logic inference —*reasoning*— [12], often trading off performance for expressivity. Despite the use of a different data model and more expressive processing, stream reasoning systems inherit the windowing mechanisms of DSMSs and CEP systems. Furthermore, since they typically exploit *all* the information in the current window to perform logical inference, they might suffer from the presence of inconsistent data, as discussed in the use cases in Section 1.

Big Data processing systems were originally designed to batch process large volumes of data on large clusters of commodity machines. Nowadays, they are shifting from pure batch computation to streaming computations [14, 13]. These systems describe the computation as a directed graph of operators. Input data elements traverse this graph and get processed one by one or in small batches.

Similar to DSMSs, operators include windows to collect portions of the streaming data. Interaction with static databases is possible, but no abstraction is provided to model and update state information.

3. A MODEL FOR EXPLICIT STATE MANAGEMENT

Figure 1 presents the model we envisage to enable explicit state management in stream processing systems. The data received from the input streams is analyzed both in the `state management` component and in the `stream processing` component. The former elaborates the input data according to a set of deployed `state management rules` to update the current state of the system, stored in the `state` repository. The latter processes the input data —together with the `state` information— to continuously produce new results for the users according to a set of deployed continuous queries or, more in general, `stream processing rules`.

Users can also query the state of the system by submitting `queries` as in traditional database systems. Finally, a `reasoning` system can extract implicit knowledge from the explicit state information to augment the answers to both stream processing rules and one-time queries. Reasoning is based on a formal description of the application domain, in the form of `ontologies`.

3.1 Case study

We now exemplify the use of our model through a case study. Let us consider a decision support tool to manage an e-commerce Web site. The managers of the Web site want to receive constant updates about the current trends of product sales, the quality of service the Web site offers —for instance, the average delay to deliver the products—, and the status of the inventory.

Traditional stream processing systems can easily satisfy these needs by periodically computing aggregates over sliding windows. For instance, the current trend of sales could be computed by summing up all the sales for each class of products over a time window selected by the user.

Nevertheless, the products and their classification change over time: new products are constantly added, new classes are created, and previous classes of products are split or merged. The information about the products and their classification is managed by a different division of the company, which updates the management whenever it is needed.

The set of available products and their classification represent background state information that the management needs to consider to correctly interpret the trends of sales.

Several approaches are possible to capture this state information. On one extreme, state can be stored in a separate database that is updated manually or semi-automatically whenever new information about the products becomes available. The stream processing system then accesses the current data in the database during processing. On the other extreme, the stream processing system might be responsible to process both the information about the sales *and* the information about the products and their classification. This approach complicates the stream processing rules, since they need to take into account heterogeneous types of data —sales and products classification— and their interaction. Furthermore, it becomes impossible to express all the processing by means of computations over sliding windows. Indeed, the system must ensure that *all* the information that builds up the most recent classification of products is taken into account, independently from the time when such information was generated.

In our model, we propose to explicitly store state information and enable the stream processing system to access that information during processing. We propose to encode the logic that updates the state based on the input streaming data into `state management rules` that the `state management` component uses to automatically handle state changes. This approach relieves the stream processing system from analyzing information related to the products and their classification, thus simplifying the `stream processing rules` that compute the selling trends.

Making the state explicit also enables the users to query such state, which would not be possible if the state information was only processed within the stream processing system. We envision the possibility to implement the `state` component as a temporal database, thus enabling the query and retrieval of both the current state and historical data. In our e-commerce case study, this enables the management to retrieve and analyze past information about products and sales to confront them with the current trends.

Finally, the `state` component can exploit domain information —for instance in the form of `ontologies`— to derive new knowledge from the explicit information it stores. For instance, in the e-commerce example, the ontology might include a taxonomy to organize the products according to different classification criteria and to automatically derive sub-classes relations.

3.2 The benefits of explicit state management

Based on the case study above, this section summarizes the benefits we see in our proposed approach.

Separation of concerns. The proposed approach decouples the management of state updates from the stream processing logic. The former is encoded in the `state management rules` while the latter is encoded in the `stream processing rules`. This enables the developers of the system to separately model these two orthogonal aspects.

Different abstractions. The separation of state management from the stream processing logic enables the adoption of separate abstractions for the two tasks. As discussed in Section 1 and in Section 2, most stream processing languages and systems are designed to express and perform continuous computations over moving windows, and this is not suitable to express state management tasks, since windows might miss some relevant state information or include contradictory data. By delegating the state management to separate rules, our approach can adopt different formalisms to express how the state is updated.

Queryable state. By making the state explicit, the proposed model enables the users to query the state on-demand, potentially referring to historical data. This would not be possible using only stream processing technologies that internally and implicitly store only the state required to execute the `stream processing rules`, and do not offer primitives to access such state. Also, queryable state can promote interoperability, since stream processing systems can expose their state and query the state of other systems.

Reuse of consolidated technologies. By clearly separating state management from stream processing, the proposed model can take advantage of consolidated technologies that are optimized for these purposes. For instance, the `stream processing` component can be easily implemented using state-of-the-art stream processing languages and systems, as presented in Section 2. Similarly, the `state` component can adopt well studied algorithms and technologies to optimize the evaluation of queries —coming both from the users and from the `stream processing` component— and to perform `reasoning` tasks.

3.3 Open research questions

This section highlights the open research questions that we are currently investigating to concretely implement the above model.

State management rules. The language used to express `state management rules` greatly influences the expressivity of the system. In the simplest case, state transitions are determined by some individual elements in the input stream. For instance, in our e-commerce use case, an individual input element might represent the new classification for a product. However, we envision more complex situations in which a state transition is determined by multiple streaming elements. We are currently investigating possible abstractions to capture these scenarios.

State representation, query, and retrieval. An open research question involves which state information to store —only the current state or also historical data—, how to represent this information —for instance, using a relational database or a key-value store—, and which language to offer for state query and retrieval.

Interaction between stream processing and state. Perhaps the most challenging question is how to define the overall semantics of the system, taking into account the possible interactions between the state —and the `state management rules`— and the `stream processing rules`. Considering the e-commerce use case, we need to define how a change in the classification of products might impact on the ongoing streaming computation.

4. RELATED WORK

The limitations of fixed count or time windows in stream processing is well known in the literature. To overcome these limitations, some approaches propose windows that are based on the content of input elements. Li et al. [10] use content-based windows to define an effective evaluation strategy for window aggregates. Similarly, predicate windows [8] define views and support view maintenance in data stream processing systems. They predicate on the content of an input element to determine whether it has to be considered as new information, or as an update (or deletion) of existing information for a given view. Frames [9] provide the developers with built-in functions to simplify the statistical analysis of data. Google Dataflow [1] proposes the concept of *session windows*, which partition a stream based on some user defined field —for instance, the identifier of a session in an e-commerce Web site—. Our model builds on similar ideas and takes a step forward by enabling the developer to express state in a more general ways, using rules that define how input elements impact on state.

Perhaps the closest work to our proposal is TEF-SPARQL [7], an extension to the SPARQL query language that distinguishes events from *facts*, where the latter are similar to the timed data elements that build the state in our model. TEF-SPARQL provides ad-hoc operators to combine facts and events and a *replace* primitive to update the set of facts. Nevertheless, TEF-SPARQL encodes the whole logic to manage and update facts within stream processing rules, whereas we separate the inference of new knowledge —including new state elements— from stream processing rules. Furthermore, we enable on-demand query of state.

5. CONCLUSIONS

Virtually all the state-of-the-art stream processing and reasoning systems rely on fixed-size windows to isolate the portions of input streams that are relevant for processing. We move from the observation that this schema is not flexible enough to effectively model several application domains and we propose a novel approach that enables the users of a stream processing system to explicitly define and modify the state of the application scenario at hand.

We believe that the approach we propose can advance the state-of-the-art in stream processing and reasoning in two orthogonal ways: on the one hand, it can ease the modeling of application scenarios in which the state of the system plays a fundamental role; on the other hand, it can simplify the processing effort by limiting the amount of streaming data that needs to be analyzed depending on the specific state of the system.

In the near future, we plan to provide a more detailed and precise formalization of our model, to implement the model into a prototype stream processing system, and to evaluate the befits of the proposed approach in terms of modeling and processing. We will consider various real world use cases with different requirements in terms of expressivity and processing complexity.

6. REFERENCES

[1] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[2] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *WWW*, pages 635–644. ACM, 2011.

[3] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16. ACM, 2002.

[5] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. Querying RDF streams with C-SPARQL. *SIGMOD Record*, 39(1):20–26, 2010.

[6] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, 44(3):15:1–15:62, 2012.

[7] S. Gao, T. Scharrenbach, J. Kietz, and A. Bernstein. Running out of bindings? integrating facts and events in linked data stream processing. In *SSN-TC/OrdRing@ISWC*, volume 1488 of *CEUR-WS Proceedings*, pages 63–74. CEUR-WS.org, 2015.

[8] T. M. Ghanem, A. K. Elmagarmid, P. Larson, and W. G. Aref. Supporting views in data stream management systems. *ACM Trans. Database Syst.*, 35(1), 2010.

[9] M. Grossniklaus, D. Maier, J. Miller, S. Moorthy, and K. Tufte. Frames: data-driven windows. In *DEBS*, pages 13–24. ACM, 2016.

[10] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD Conference*, pages 311–322. ACM, 2005.

[11] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2001.

[12] A. Margara, J. Urbani, F. van Harmelen, and H. E. Bal. Streaming the web: Reasoning over dynamic data. *J. Web Sem.*, 25:24–44, 2014.

[13] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

[14] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438. ACM, 2013.

Stability notions in synthetic graph generation: a preliminary study

Wilco van Leeuwen
George H.L. Fletcher
Nikolay Yakovets
Eindhoven University of Technology
Eindhoven, The Netherlands
{w.j.v.leeuwen@student., g.h.l.fletcher@, hush@}tue.nl

Angela Bonifati
University of Lyon 1
Lyon, France
angela.bonifati@univ-lyon1.fr

ABSTRACT

With the rise in adoption of *massive* graph data, it becomes increasingly important to design graph processing algorithms which have *predictable* behavior as the graph *scales*. This work presents an initial study of *stability* in the context of a schema-driven synthetic graph generation. Specifically, we study the design of algorithms which generate high-quality sequences of graph instances. Some desirable features of these sequences include *monotonic containment* of graph instances as they grow in size and *consistency* of structural properties across the sequence. Such stability features are important in understanding and explaining the scalability of many graph algorithms which have cross-instance dependencies (e.g., solutions for role detection in dynamic networks and graph query processing). We implement a preliminary approach in the recently proposed open-source synthetic graph generator gMark and demonstrate its viability in generating stable sequences of graphs.

1. INTRODUCTION

Rising adoption of massive graph data (e.g., social networks, WWW, biological data) necessitates the development of algorithms which are able to handle the vast amounts of information stored in these datasets. Synthetic graph generators are a valuable tool for investigating the performance of graph processing algorithms since it allows to generate a sequence (or, a *family*) of graph instances of increasing sizes. For performance studies to be useful and reproducible, it is important to ensure the *quality* of the generated instances themselves and the instance family, as a whole. The simplest quality measure of the instance is its size. In some cases (e.g., in benchmarking of naive graph serialization), generation of different-sized independent random graph instances is sufficient to demonstrate the performance characteristics of the algorithm. However, in other cases (e.g., in benchmarking of query processing engines), stronger quality guarantees are desired. For example, in addition to graph size, a user-defined graph *schema* might include the enumeration of node and edge labels along with their proportions in the generated graph instances. Furthermore, in- and out-degree distributions can be defined on top of the constraints on source-target node pairs in a graph. Graph generation approaches that produce graph instances which conform to such extended schema are called *schema-driven.*

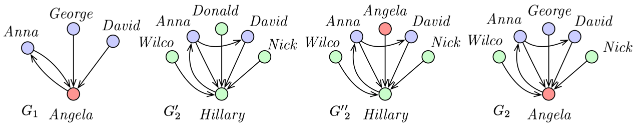


Figure 1: Node-level changes between instances. In graph instances G_1 and G_2, node *Angela* is a “leader” with relatively high in-degree. In G'_2, however, *Angela* does not appear at all, while in G''_2, she is merely a “follower.”

In benchmarking of graph algorithms it is often important to control how relationships involving specific nodes *evolve* between graph instances in the family. Ideally, given a particular node, its structural properties should be stable across the entire sequence of generated graph instances. For example, queries which mention constants are common in the design of benchmarks, as they allow fine-tuned control of query selectivity and run-time behavior [4, 5, 8, 9, 12]. As another example, in the study of solutions for role detection in social networks, it is often desirable that structural features of individual nodes (i.e., individual actors in the network) are stable as the network grows in size.

To concretely illustrate stability, consider the graph instances shown in Figure 1. Here, G_2, G'_2, and G''_2 have the same size and are all larger than G_1. Suppose that a given schema (S) models a typical behavior of follows edges in a social network. This schema defines the in-distribution of all edges in the graph to be Zipfian and all nodes to be of type Person. Observe that each of the instances G_1, G'_2, G''_2, and G_2 satisfy S. Suppose, we fix a specific node in G labeled *Angela* and trace its behavior across instances in Figure 1. Specifically, *Angela* is a “leader” (followed by many) in G_1 and G_2, but is not present in G'_2 or is not a leader in G''_2. Hence, using instance families $\{G_1, G'_2\}$ and $\{G_1, G''_2\}$ would lead to inconsistent results when studying the behavior of an

Series ISSN: 2367-2005

10.5441/002/edbt.2017.51

algorithm which depends on *Angela*, e.g., in a performance study of a query evaluation engine using a benchmark query which specifically mentions *Angela*.

State of the art. The study of solutions for controlled generation of synthetic database instances has a long history in the data management community [2, 10]. In the domain of graph databases, synthetic generation of realistic graphs is currently a topic of intense investigation [1,4,5,7,8,13,14]. In this context, quality metrics on individual synthetic graphs (with respect to a given real graph or a given schema) have been proposed, e.g., [3,7,11,13,14]. To our knowledge, however, there has been no prior study of stability across **sequences** of graphs in synthetic instance generation.

Contributions. Motivated by these observations, in this paper we initiate the study of stability in synthetic graph generation. We consider basic properties of instance families which are, to the best of our knowledge, not satisfied by current schema-driven graph generators. We present the preliminary design of a scalable solution for producing instance families which are *stable* with respect to edge-type degree distributions defined by a given schema (e.g., for the Zipfian distribution in the family $\{G_1, G_2\}$ of Figure 1).

The detailed structure and contributions of this paper are as follows. We define two basic desirable properties of generated graph instance families (§2.1) and design a novel measure of distribution stability for a given family (§2.1). We then present an algorithm for generation of stable instance families and analyze its complexity (§2.2). We implement our approach in the state-of-the-art open-source gMark graph generator [4, 5], and demonstrate that our solution is scalable (§3.1) and produces instance families which are significantly more stable than the original gMark instance generator (§3.2).

2. STABLE GENERATION

2.1 Preliminaries

We study the following problem, on finite directed edge-labeled graphs. Given a finite sequence of positive integers $n_1, \ldots, n_k$ such that $n_i < n_{i+1}$, for $1 \leq i \leq k$, generate a sequence of graphs $\mathcal{F} = (G_1, \ldots, G_k)$ such that $|nodes(G_i)| = n_i$, for $1 \leq i \leq k$, where $nodes(G)$ denotes the node set of graph G and $|A|$ denotes the size of set A. If we are additionally given a graph schema S as input, we further require that each G_i is a valid instance of S.

In this initial study, we consider the following desirable properties of generated graph sequences:

- *Monotonicity.* It holds that $edges(G_i) \subseteq edges(G_{i+1})$, for $1 \leq i \leq k$, where $edges(G)$ denotes the edge set of graph G.
- *Distribution stability.* If the degree structure of an edge type follows a fixed distribution (e.g., edges labeled follows in a social network have a Zipfian in-distribution), then the position of nodes in the distribution is stable throughout the graph sequence $\mathcal{F}$.

The degree (deg) of a node needs to be stable in its distribution in all graphs of a given sequence. To capture this, we define the *rank* of a node n in graph G as:

$$rank(n, G) \quad = \quad \frac{deg(n)}{\max_{n \in nodes(G)}(deg(n))}$$

where the value of $rank(n, G)$ ranges from 0 to 1. Letting σ_n denote the standard deviation of the rank of n over all instances in $\mathcal{F}$, i.e., the standard deviation of the set $\{rank(n, G) \mid G \in \mathcal{F}\}$, we define the *stability* of n in $\mathcal{F}$ as:

$$stability(n, \mathcal{F}) \quad = \quad 1 - 2\sigma_n.$$

A $stability(n, \mathcal{F}) = 1$ indicates that the rank of n never changes, whereas $stability(n, \mathcal{F}) = 0$ means that n is completely unstable, with respect to degree structure.

2.2 Generation Approach

The original gMark graph generator (gMarkGraphGen) does not satisfy monotonicity of graph generation nor does it guarantee distribution stability; the time complexity of generation is linear in the number of nodes of the graph instance [5]. We next propose an algorithm, MonStaGen, that generates instance families which are both monotonic and stable with respect to edge-type degree distributions defined by a given schema. An analysis of this algorithm, however, shows that the satisfaction of these properties comes at the expense of increased time complexity ($O(n \log n)$, where n is the number of nodes).

MonStaGen separately calls a procedure (Algorithm 1) for each edge type in a given schema. As a consequence, the generation of subgraphs that correspond to each edge type can be executed in parallel. Consider the graph of a single edge type as a bipartite graph with the two disjoint sets S and T. Set S represents all the subject nodes of the edge type, whereas set T represents all the object nodes of the edge type. Graph generation proceeds by iteratively adding edges from nodes in S to nodes in T.

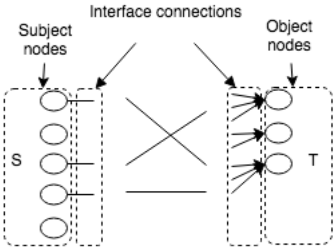


Figure 2: Generating the edges of one edge-type with five subject nodes and three object nodes.

Figure 2 shows an example of the generation of one edge-type with five subject nodes (elements of set S) and three object nodes (elements of set T). This figure also introduces our concept of *interface connections*. We call the connections (i.e., edges) that a node can potentially receive the *interface connections (ICs)* of this node. Whenever a new node is added to the graph, the degree distributions in a given schema determine the number of ICs of this node depending on whether it is a subject (out-degree) or an object (in-degree). Whenever a new edge is added, the participating ICs for its subject and object nodes are closed.

Generated subject nodes, object nodes, and edges are cached for subsequent processing by functions *addSubjectNodes* and *addObjectNodes*. Function *addEdge* decrements the amount of open ICs of the subject node and the object node by one. Observe that, by construction, the generated sequence of graphs satisfies the monotonicity property. Next, we discuss the immediate challenges that need to be

Algorithm 1 processEdgeType(edgeType, graph, probability p, #subjectNodes, #objectNodes)

```
graph.addSubjectNodes(#subjectNodes)
graph.addObjectNodes(#objectNodes)

if edgeType.subjectNodes is scalable xor edgeType.objectNodes is scalable then
    updateICsForNonScalableNodes()
end if
if in- or out-degree distribution is Zipfian then
    updateICsForNodesWithZipfianDistribution()
end if

vector $v_{src}$, $v_{trg}$
for subject in graph.subjects do
    for 1:subject.openICs do
        $v_{src}$.add(subject)
    end for
end for
for object in graph.objects do
    for 1:object.openICs do
        $v_{trg}$.add(object)
    end for
end for
shuffle($v_{src}$); shuffle($v_{trg}$)

for i $\in$ 1:min($v_{src}$.length, $v_{trg}$.length) do
    with probability $p$, graph.addEdge($v_{src}$[i], $v_{trg}$[i], edgeType.predicate)
end for

return graph
```

met during schema-driven stable graph generation.

Subject-to-object scalability mismatch. Consider the edge-type *Persons live in a City*, where the number of persons scale with the graph size, i.e. the subject nodes are scalable, and the number of cities is fixed. When the number of ICs are fixed for the non-scalable object nodes, we will reach a point where new edges cannot be generated anymore. This is because the total number of ICs in the *City* nodes will remain the same when growing the graph, whereas the total number of ICs in the *Person* nodes will grow. It is still possible to grow the graph after this point, resulting in more *Person* nodes with new ICs. However, these ICs can never be used anymore, since all object nodes are not able to receive any connection and new objects cannot can be added. This problem is solved by updating the number of ICs of the non-scalable nodes.

Skewed distributions. During the construction of the new graph instance, the ICs of the nodes which participate in a Zipfian distribution need to be updated to ensure that nodes with a very high degree will be able to continue to receive more connections.

We proceed by creating separate vectors for subject and object nodes with as many entries for each node as it has assigned ICs. We then randomly shuffle these two vectors. Finally, with probability p, we add each edge from the source vector to the target vector. Here, p ensures the balance between the connectedness of the new instance with later instances in the sequence, on one hand, and the quality of the degree distributions, on the other.

3. EMPIRICAL STUDY

In this section, we report the results of three experimental studies of our approach. In all of the experiments, we investigated the generation of graphs with skewed (Zipfian) degree distributions. We selected Zipfian distribution parameter of 2.5, as it corresponds to structure found in many real-world graphs [6]. The input parameter p to MonStaGen (Algorithm 1) was set to 0.97.

3.1 Run-time

We design two experiments to demonstrate the running time of MonStaGen compared to gMarkGraphGen. As noted in §2.2, the complexity of MonStaGen is slightly higher ($O(n \log n)$ vs. $O(n)$). Figures 3a and 3b show the difference in the running time for both approaches. In our first experiment (Figure 3a), we benchmark the generation of an instance family which consists of a single graph instance of increasing size. Here, gMarkGraphGen slightly outperforms MonStaGen due to increased complexity required by MonStaGen to satisfy the monotonicity and stability properties of instance families.

In our second experiment (Figure 3b), we generate families of increasing size with ten graph instances each $\mathcal{F} = G_1, \ldots, G_{10}$. In MonStaGen, for producing graph G_{i+1}, the previously generated graph G_i is used, instead of generating the whole graph from scratch. We recall that sequences generated by gMarkGraphGen do not satisfy the monotonicity property.

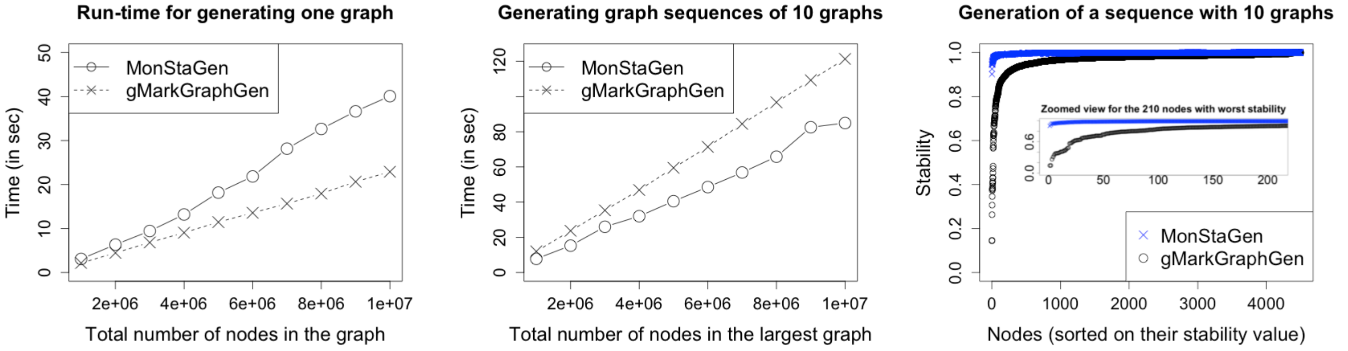
In Figure 3b, the *x-axis* corresponds to the total number of nodes in the largest graph (G_{10}) in the generated instance. We set the number of nodes in graph G_i to a fraction $\frac{i}{10}x$, where x is the total number of nodes in graph G_{10}.

As expected, MonStaGen is slower when generating a single graph, whereas it supersedes gMarkGraphGen when generating multiple graphs. This means that MonStaGen scales with the number of graphs in the sequence, while gMarkGraphGen scales with the number of nodes in the graphs. This behavior is quite interesting and leaves open the user's choice of which generator to employ in a given application.

3.2 Stability of nodes in degree distribution

We next consider generation of a graph sequence $\mathcal{F} = (G_1, \ldots, G_{10})$, where the total number of nodes of G_i is $i \cdot 1000$, for $1 \leq i \leq 10$. Figure 3c shows the stability of nodes of a single edge type with Zipfian degree distribution, as noted above, and $0.5n$ subject nodes, where n is the total number of nodes in the graph.

The nodes added in the last graph are not taken into account, because they will always have a stability of 1. The total number of subject nodes taken into account is $0.5 \cdot (10 - 1) \cdot 1000 = 4500$. The stability value for all of these nodes, calculated and sorted on such a value, is illustrated in Figure 3c. We can see that the stability of nodes in MonStaGen is significantly higher than gMarkGraphGen. The long tail, which indicates a high number of nodes with a high stability, is the effect of a Zipfian distribution, where a node has a very high probability of having a very low degree. This means that many nodes will have a low degree in all the graphs of the sequence, resulting in a high stability.



(a) Average run-time of the generation of single graph instances. Each data point is the average of four runs, after dropping the highest and lowest values.

(b) Average run-time of the generation of a graph sequence with 10 graphs with equal increments. Averages taken as in (a).

(c) Stability comparison of all the nodes in a Zipfian(2.5) degree distribution. A zoom of the 210 nodes with worst stability is shown inside the plot.

Figure 3: Experimental comparisons of gMarkGraphGen and MonStaGen.

If we consider the stability of nodes, we see that the minimal stability in MonStaGen is 0.9020 and the median value is 0.9988. In the graphs generated by gMarkGraphGen, 210 nodes that have a lower stability value than our worst-case 0.9020 and 4393 nodes that have lower stability than our median 0.9988. In other words, with gMarkGraphGen, over 97% of the nodes are more unstable than the median value of those nodes generated with MonStaGen. The inner plot of Figure 3c shows a zoom-in on the 210 most unstable nodes.

4. LOOKING AHEAD

In this preliminary study, we have highlighted the utility of properties, such as stability, characterizing graph sequences rather than individual graphs. We have proposed a first algorithm for generating graph sequences conforming to a schema which satisfy the monotonicity property and are significantly more stable than those generated using gMark, a state-of-the-art synthetic graph generator. To our knowledge, ours is the first schema-driven synthetic graph generator having these properties.

Our study opens up several directions for future work. A major direction is to establish further stability properties occurring in real-world graphs, and extending our graph generation algorithm to capture these (e.g., stability with respect to node-centrality measures). Further, in addition to edge insertion, we plan to consider other natural aspects of graph evolution such as edge deletion, batch updates, and temporal dynamics. Finally, we would like to incorporate our solutions in the open-source gMark framework.[1]

5. REFERENCES

[1] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of RDF data management systems. In *ISWC*, pages 197–212, Riva del Garda, Italy, 2014.

[2] A. Arasu, R. Kaushik, and J. Li. Data generation using declarative constraints. In *SIGMOD*, pages 685–696, Athens, Greece, 2011.

[3] M. Arenas, G. I. Diaz, A. Fokoue, A. Kementsietsidis, and K. Srinivas. A principled approach to bridging the gap between graph data and their schemas. *PVLDB*, 7(8):601–612, 2014.

[4] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. Generating flexible workloads for graph databases. *PVLDB*, 9(13):1447–1460, 2016.

[5] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. gMark: Schema-driven generation of graphs and queries. *IEEE Trans. Knowl. Data Eng.*, in press, 2017.

[6] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.

[7] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *SIGMOD*, pages 145–156, Athens, Greece, 2011.

[8] O. Erling et al. The LDBC social network benchmark: Interactive workload. In *SIGMOD*, pages 619–630, Melbourne, 2015.

[9] A. Gubichev and P. A. Boncz. Parameter curation for benchmark queries. In *TPCTC*, pages 113–129, Hangzhou, China, 2014.

[10] E. Lo, N. Cheng, W. W. K. Lin, W. Hon, and B. Choi. MyBenchmark: generating databases for query workloads. *VLDB J.*, 23(6):895–913, 2014.

[11] Y. Luo, G. H. L. Fletcher, J. Hidders, P. D. Bra, and Y. Wu. Regularities and dynamics in bisimulation reductions of big graphs. In *GRADES*, page 13, New York, NY, 2013.

[12] C. Mishra, N. Koudas, and C. Zuzarte. Generating targeted queries for database testing. In *SIGMOD*, pages 499–510, Vancouver, BC, 2008.

[13] S. Qiao and Z. M. Özsoyoglu. Rbench: Application-specific RDF benchmarking. In *SIGMOD*, pages 1825–1838, Melbourne, 2015.

[14] J. W. Zhang and Y. C. Tay. GSCALER: synthetically scaling a given graph. In *EDBT*, pages 53–64, Bordeaux, 2016.

[1] https://github.com/graphMark/gmark

Big Spatial Data Processing Frameworks: Feature and Performance Evaluation

- Experiments & Analyses -

Stefan Hagedorn
TU Ilmenau, Germany
stefan.hagedorn@tu-ilmenau.de

Philipp Götze
TU Ilmenau, Germany
philipp.goetze@tu-ilmenau.de

Kai-Uwe Sattler
TU Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT

Nowadays, a vast amount of data is generated and collected every moment and often, this data has a spatial and/or temporal aspect. To analyze the massive data sets, big data platforms like Apache Hadoop MapReduce and Apache Spark emerged and extensions that take the spatial characteristics into account were created for them. In this paper, we analyze and compare existing solutions for spatial data processing on Hadoop and Spark. In our comparison, we investigate their features as well as their performances in a micro benchmark for spatial filter and join queries. Based on the results and our experiences with these frameworks, we outline the requirements for a general spatio-temporal benchmark for Big Spatial Data processing platforms and sketch first solutions to the identified problems.

1. INTRODUCTION

In the Big Data era, almost every piece of information produced is also stored and used for analyses. The produced information can be of any kind: plain web server logs, sensor readings from home or environment monitoring, (mobile) location-aware devices that periodically report their position, complex entities in Open Data sets like Wikipedia/WikiData, or structured event information extracted from news articles and other text sources. Often these types have at least two features in common: a time and a location component. For scalable processing of large datasets, data parallel architectures like Hadoop MapReduce and Apache Spark have been introduced and have widely been accepted. However, their general data model does not take spatial or temporal relations of the data items into account and therefore cannot efficiently answer spatial, temporal, or spatio-temporal queries.

In this paper, we present results of an experimental study comparing existing solutions for spatial data processing on Apache Hadoop and Apache Spark. Particularly, we consider the Hadoop extensions Hadoop-GIS [1] and SpatialHadoop [2] as well as the Spark-based systems SpatialSpark [3], GeoSpark [4], and our own implementation STARK[1]. We investigate their features and also compare their performance in a micro benchmark for spatial filter and join queries. Finally, we will conclude with an outlook to a general spatial and spatio-temporal benchmark.

2. EXISTING SOLUTIONS FOR BIG SPATIAL DATA PROCESSING

The first approach to implement spatial operations as an extension for Hadoop MapReduce is SpatialHadoop [2, 5]. It is built on top of Hadoop and provides spatial operators for range queries, k nearest neighbors, and joins that can be integrated into any Hadoop MapReduce program. Furthermore, spatial partitioning and indexing is available, too.

Another approach that extends the plain Hadoop MapReduce framework with spatial operators is Hadoop-GIS [1]. Similarly to SpatialHadoop, Hadoop-GIS utilizes a two-level indexing: a global partition indexing and an optional local spatial indexing. The query processing engine RESQUE (written in C++), uses these indexes to identify partitions to load and to speed up processing the required partitions. The RESQUE engine provides spatial operators for filters and joins and is integrated into the Hive ecosystem.

While Hadoop is a very fault tolerant environment for parallel execution, writing all intermediate results to disk makes the execution slow. Hence, the in-memory execution model of Spark became very popular as it reduces the execution time drastically, compared to MapReduce jobs. Currently, there are two systems that implement spatial operators for Spark: GeoSpark and SpatialSpark.

GeoSpark [4, 6] is a Java implementation that comes with four different RDD types: `PointRDD`, `RectangleRDD`, `PolygonRDD`, and `CircleRDD`. These special RDDs internally maintain a plain Spark RDD that contains elements of the respective type, i.e., points, rectangles, polygons, and circles. GeoSpark supports k nearest neighbor queries, range queries, as well as joins and each of these queries can be executed with or without using an index.

The main goal of the SpatialSpark approach described in [3] is to provide a parallel join technique for large spatial data sets. For this, they focus on data processing on parallel hardware like multi-core CPUs and GPUs. SpatialSpark implements a broadcast join, where the right relation is read into memory and distributed to all workers. If the relation is

[1] https://github.com/dbis-ilm/stark

Series ISSN: 2367-2005

10.5441/002/edbt.2017.52

Table 1: Feature comparison of Hadoop- and Spark-based Big Spatial Data processing platforms

	Hadoop-GIS	SpatialHadoop	GeoSpark	SpatialSpark	STARK
Query Language / DSL	✓	✓	×	×	✓
Spatio-Temporal Data	×	×	×	×	✓
Spatial Partitioning	✓	✓	✓	✓	✓
Indexing	✓	✓	✓	✓	✓
Persistent Indexes	✓	✓	×	✓	✓
Filter				no partitioning	
Contains	✓	✓	✓	(✓- w/o Index)	✓
ContainedBy	✓	✓	×	(✓- w/o Index)	✓
Intersects	✓	✓	✓	(✓- w/ Index)	✓
WithinDistance	✓	✓	×	(✓- w/o Index)	✓
Join	✓	✓	(✓- pred. limit.)	(✓- returns IDs)	✓
k Nearest Neighbors	✓	✓	✓	×	✓
Clustering	×	×	×	×	✓

too big for main memory, a spatial partitioning and indexing is utilized [7].

In the next sections, we will have a deeper look into the supported features and limitations of the mentioned systems and will also compare the performances of their operators.

3. FEATURE COMPARISON

The DE-9IM [8] defines all possible relations between two spatial objects and the Open Geospatial Consortium released a standard for spatial data types and operations, which is implemented in many spatial DBMS. In Table 1 we compare the five engines based on a subset of these standards and additionally include aspects like query language, spatial partitioning, indexing, and data analysis operators.

Query Language.

Hadoop-GIS is integrated into Hive and implements the SQL/Spatial MM standard and includes a complete set of predicates, according to DE-9IM that can be used with filter and join operators. For SpatialHadoop the authors introduced the Pigeon [9] language that extends Pig Latin with spatial functions. In Pigeon, fields of type `bytearray` are implicitly converted to a `geometry` type when needed. SpatialHadoop programs can also be written as plain MapReduce programs, but as we did not find any documentation we found it hard to set the correct classes to parse spatial data and set parameters for a simple range query correctly. Furthermore, SpatialHadoop provides a command line script that can be used to run a single query/join.

GeoSpark can be used via its Java API, which however does not integrate well into the Spark API. Unlike in Spark where transformations are defined as methods on an RDD, in GeoSpark users have to create extra instances of, e.g., `PointRDD` and pass in the base `JavaRDD`. For the operations, again a new instance of the operator class, e.g., `RangeQuery` has to be created which accepts the GeoSpark-RDD to work on. This makes it tedious to write complex programs and to represent a data flow. The main drawbacks of GeoSpark are these special RDDs, which can only hold geometries of one certain type (points in `PointRDD`, polygons in `PolygonRDD`, ...). On the one hand, this makes it impossible to load a dataset that contains different geometry types in one column and, on the other hand, all other columns are removed when putting the data into these spatial RDDs. This also means that it is not possible to process the data in subsequent steps since related columns such as an ID are not available anymore.

It seems that SpatialSpark should be used only via the command line and run single queries/operations. However, the internal Scala classes can be used in other programs as well, although there is very little documentation.

STARK provides an integrated DSL (domain specific language) for spatio-temporal query processing that seamlessly integrates into any (Scala) Spark program. Spatial Joins and filters can be called directly as transformations on standard RDDs. Additionally, it allows defining custom distance functions and predicates for its operators.

To the best of our knowledge from the found literature and provided documentation, only SpatialHadoop and STARK are able to also process temporal or spatio-temporal data.

Spatial Partitioning and Indexing.

Hadoop-GIS comes with a recursive grid partitioning and a global index (R-tree, R*-tree). This index is stored in the HDFS and used to identify partitions that need to be loaded, i.e., that contribute to the result. Furthermore, objects within a partition (tile) can be indexed as well on demand. SpatialHadoop also employs two index levels: on a global level an index partitions data across all nodes while a local index organizes data inside each partition. The indexes hold a copy of the data to avoid random HDFS lookups [10]. The number of generated partitions is calculated depending on the input file size, the HDFS block size, and an overhead ratio. These indexes are used on read to eliminate records that do not contribute to the final result. As index structures, SpatialHadoop supports grid files, R-tree, and R+-trees.

GeoSpark comes with several partitioning techniques: R-tree partitioning as well as Voronoi, Hilbert, and fixed grid partitioning. As described in [4], it supports R-trees and quadtrees to create an ad hoc index on the RDDs. However, during the evaluation, we found that choosing quadtrees is not possible and respective settings are ignored. Persisting indexes in GeoSpark seems not to be possible, since there is no way to load and index or assign it to an RDD.

SpatialSpark includes a fixed grid partitioning, binary space partitioning (BSP), as well as tile partitioning. Indexes have to be created and written to disk/HDFS before they can be used within a program, i.e., there is no possibility for live on-the-fly indexing.

STARK includes a fixed grid partitioner as well as a cost BSP, which ensures partitions with almost same cost (amount of data items). Indexes in STARK can be created on-the-fly within a program and can also be materialized for later use. However, in STARK an index that should be materialized can also be used within the same program, while in the other frameworks, this index has to be created in an extra run and then has to be explicitly loaded. Currently, STARK uses only R-trees for indexing.

Spatial Filter & Join Operators.

Hadoop-GIS and SpatialHadoop are DE-9IM compatible and spatial filter and join operators can be used with many predicates.

GeoSpark only provides a *contains* and *intersects* predicate for spatial filters. For spatial joins only *contains* and, for joining two point data sets, *withinDistance* is supported. For joins spatial partitioning is obligatory, but indexing cannot be used.

SpatialSpark supports spatial filter queries with the predicates *contains*, *within* (*containedBy*), and *withinDistance*. However, a spatial partitioning cannot be applied in combination with the filter operator. When querying a persistent index for these range queries the *intersects* predicate is compulsorily used. Internally, they expect RDDs with an ID and a geometry object, which are processed when calling the specific query object (like `RangeQuery` or `BroadcastSpatialJoin`). SpatialSpark does not allow other payload fields but the ID and, furthermore, the result of a join returns only the matched pairs IDs, which requires additional joins afterwards to retrieve the complete tuple in the application.

STARK includes a wide range of spatial predicates (that can also be used for spatio-temporal data) which are applicable for filter and join. While other systems neglect payload data and only work with IDs and geometries, STARK keeps any payload data throughout all operations.

Other Data Analysis Operators.

All considered frameworks support a k-nearest neighbor operator, except SpatialSpark. However, they provide a 1-nearest-neighbor join predicate. A clustering operator is only available in STARK, which implements DBSCAN.

4. PERFORMANCE EVALUATION

In the following experiments, we focus on a micro benchmark comparing the execution times for single operators with different settings. The benchmarks are executed on our cluster of 16 nodes with an Intel Core i5 processor and 16 GB RAM on each node. The nodes are interconnected with a 1 Gbit/s network. The Spark jobs are executed with 32 executors and 2 cores for each executor. The data generator, test programs, settings, as well as more experiments and results are available on GitHub[2] . To run our experiments, we first had to fix issues in GeoSpark[3]. The most important problem was that operations that use an index for querying returned the candidate set returned by the index (R-tree). We added the candidate pruning step to obtain the correct results. Furthermore, the *contains* predicate was actually a *containedBy* (the operands were swapped).

The first experiment executes a spatial filter operator over a 50,000,000 polygon data set (880 GB, uniform distribution) with a *contains* predicate to find those polygons that contain a given query point. We used all available spatial partitioners of each framework and executed the operation without indexing as well as with live (on-the-fly) index creation, if possible. In this experiment, STARK performed best with only 47 (BSP, live index) or 52 seconds (BSP, no index). SpatialSpark is very limited in its usability as a spatial partitioning is not allowed in combination with a filter. The run without spatial partitioning and indexing took 3866 seconds (more than 1 hour). GeoSpark needed 1237 seconds (20 minutes) without partitioning but was not able to process this data set at all with a spatial partitioner and crashed after several hours for each partitioner. While investigating this problem we executed the program on a smaller polygon data set with 1,000,000 entries (17,6 GB). In the best case, it took 54 seconds with Hilbert partitioning. That is the same time that STARK needed to process 880 GB. For SpatialHadoop (as a representative of the Hadoop based systems) we used their command line program, but were not able to receive a correct result: The program finished after 39 minutes with zero results. The problem is that a point query option is not available and so we provided a point as query range. A visualization of the execution times along with a more detailed analysis that includes the overhead of the partitioning can be found in our GitHub repository[2] .

Our next experiment examines the influence of the query range size to the execution time. For this, we used a point data set with 50,000,000 points and executed a filter operator with a *containedBy* predicate to find all points contained by a given polygon. While the data set has a value range of [-180, 180] for `x` coordinates and [-90,90] for `y`, we execute the filter with 5 different squares created as polygons. These squares have the side lengths: 1, 5, 10, 50, and 100. Additionally, there is a query range that covers the complete data space. Figures 1 to 3 show the execution times for all partitioner/indexing combinations. Partitioners that require setting the number of partitions in advance all use the same amount (8100). This shows the impact of the pruning step that the frameworks can take. If the query region is small, only a single or very few partitions may contain result objects and other partitions do not have to be checked. STARK makes use of this partition pruning approach where ever possible and thus, is able to outperform the others that do not seem to perform this action as they need the same time for all six queries.

In the last experiment that we show here, we analyzed the spatial join operation on two point data sets (1,000,000 points, uniform distribution) that finds equal points (same exact location) in the two data sets. Figure 4 shows the result for the Spark based frameworks with their best partitioner for the join with and without using an index. We were not able to perform this test with SpatialHadoop as the command line program crashed with an error and we did not find any helpful documentation. For GeoSpark and SpatialSpark the same partitioner performed best in both cases. However, GeoSpark has a bug for Grid and R-tree partitioning as in the final result 1 and ca. 10,000 tuples respectively were missing (we also encountered different result sizes for in each repetition of the experiment). It can also be seen that for these frameworks, one cannot benefit from

[2] https://github.com/dbis-ilm/spatialbm

[3] We further had to use version 0.3 as the newer version 0.3.2 crashed with out-of-memory errors for the same settings.

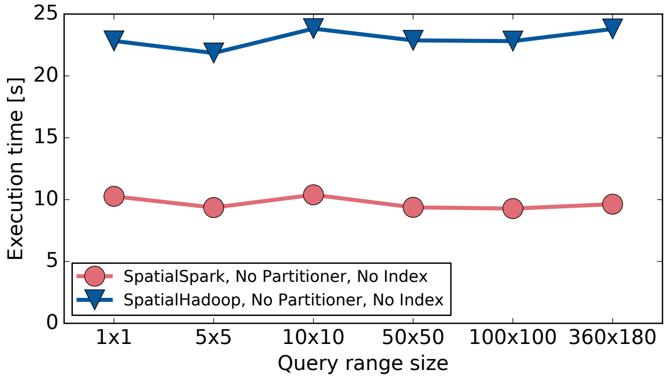


Figure 1: Exec. times for different range query sizes for **SpatialHadoop** & **SpatialSpark**

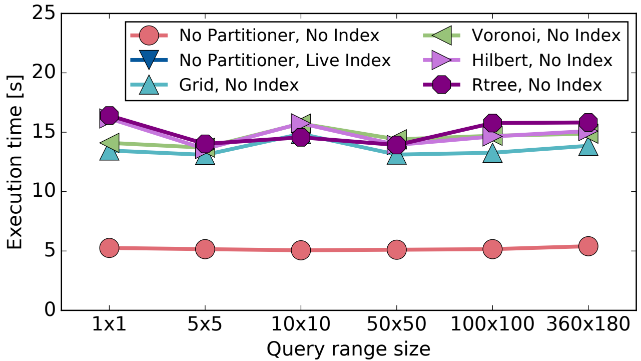


Figure 2: Exec. times for different range query sizes for **GeoSpark**. *No Partitioner, Live Index* is out of range (40 sec) for all queries.

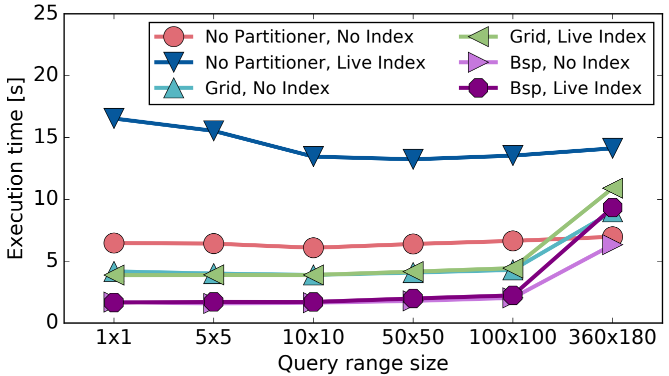


Figure 3: Exec. times for different range query sizes for **STARK**.

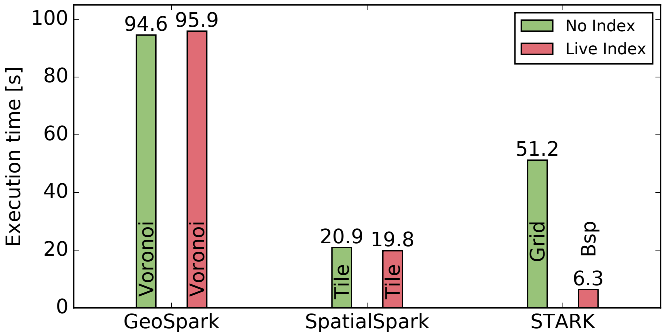


Figure 4: Exec. times for spatial join queries.

live indexing. This maybe because the speedup of querying the point index is not big enough to compensate the time required to build the index. For STARK, without using an index the Grid partitioner performed best, but was slower than SpatialSpark. With live indexing, however, the BSP was best and outperformed the others. The reason here may be that the BSP created partitions with an equal number of elements and thus equal workload on the executors.

5. CONCLUSION

In this paper, we introduced Hadoop and Spark based engines that allow processing Big Spatial Data. As the results of our feature comparison and micro benchmark show, they all differ in supported operations as well as in implementation and thus, in performance. However, the performed micro benchmark should just be a starting point for a more exhaustive spatial benchmark. For this, a more flexible data generator is needed to easily create clustered data of different sizes. Furthermore, a good benchmark should contain micro benchmark tests, as shown in the previous section, as well as a macro benchmark performing real world queries. The macro benchmark is needed to (1) evaluate the real performance and (2) to compare the functionality and usability of the system. Although the queries may be formulated in natural text leaving the task of the implementation to the authors and developers of an engine, we believe that the Pigeon [9] extension for Pig Latin in combination with our Piglet [11] for code generation will provide a good and portable user interface for such a benchmark.

Acknowledgments.

This work was partially funded by the German Research Foundation (DFG) under grant no. SA782/22

6. REFERENCES

[1] A. Aji, F. Wang *et al.*, "Hadoop-GIS: A High Performance Spatial Data Warehousing System over Mapreduce," *VLDB*, pp. 1009–1020, 2013.

[2] A. Eldawy and M. F. Mokbel, "A Demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data," in *VLDB*, 2013.

[3] S. You, J. Zhang, and L. Gruenwald, "Large-Scale Spatial Join Query Processing in Cloud," in *ICDE*, 2015.

[4] J. Yu, J. Wu, and M. Sarwat, "GeoSpark: A Cluster Computing Framework for Processing Large-Scale Spatial Data." SIGSPATIAL, 2015, p. 70.

[5] Eldawy and Mokbel, "SpatialHadoop: A MapReduce Framework for Spatial Data," in *ICDE*, 2015.

[6] J. Yu, J. Wu, and M. Sarwat, "A Demonstration of GeoSpark: A Cluster Computing Framework for Processing Big Spatial Data," *ICDE*, 2016.

[7] S. You, K. Gorlo *et al.*, "Big Spatial Data Processing using Spark," https://git.io/vXiMd, accessed Nov. 14, 2016.

[8] M. J. Egenhofer and R. D. Franzosa, "Point-set topological spatial relations," *IJGIS*, 1991.

[9] A. Eldawy and M. F. Mokbel, "Pigeon: A spatial MapReduce language," in *ICDE*, 2014.

[10] R. T. Whitman, M. B. Park *et al.*, "Spatial Indexing and Analytics on Hadoop." SIGSPATIAL, 2014.

[11] S. Hagedorn and K.-U. Sattler, "Piglet: Interactive and Platform Transparent Analytics for RDF & Dynamic Data," in *WWW*, April 2016, pp. 187–190.

Implementation and Evaluation of Genome Type Processing for Disease-Causal Gene Studies on DBMS

Yoshifumi Ujibashi
Fujitsu Laboratories Ltd.
Kawasaki, Japan
ujibashi@jp.fujitsu.com

Motoyuki Kawaba
Fujitsu Laboratories Ltd.
Kawasaki, Japan
kawaba@jp.fujitsu.com

Lilian Harada
Fujitsu Laboratories Ltd.
Kawasaki, Japan
harada.lilian@jp.fujitsu.com

ABSTRACT

Recent development of next generation sequencer (NGS) and algorithms for genomic analysis are contributing to the understanding of human genetic variation and thus to personalized medicine. In leveraging this genomic data, the difficult task of finding out the genes relevant to dedicated phenotypes, e.g., disease-causal gene analysis, is becoming increasingly important. In a previous work, we have introduced a user defined function called "genome type" into PostgreSQL open source relational database management system (RDBMS) to accelerate genetic analysis with the aid of an efficient data structure in which all the genotypes are packed into one record.

However, there are still some challenges to be addressed in order to efficiently implement the proposed genome type when using real data. One problem is that, although the majority of variants are composed of three types of genotypes, this number is not fixed and can be highly skewed. Another problem is that the amount of genomic data necessary for accurate association analysis is huge and speed-up is necessary to make an iterative analysis feasible.

To solve these problems, we developed a new method which efficiently stores and processes variants of variable sizes. We also applied query parallelization techniques and exploited instruction level parallelization (SIMD) on Intel Xeon processor. Our performance evaluation shows that the processing of very large scale genomic data can be reduced to some few seconds.

1. INTRODUCTION

Due to recent technological innovations on genome sequencing and analysis, the speed and cost of acquiring genome information have drastically reduced and thus huge amounts of genome information has been collected. Human DNA consists of 3 billion DNA sequences and is represented by a sequence of four base AGCTs. Currently, it is said that there are tens of millions of variations such as SNP (Single nucleotide polymorphism), INDEL (insert-deletion), CNV (Copy Number Variation) that derive individual differences in their phenotypes. In recent years, disease-causal gene studies such as cohort studies and case-control studies that are based on statistical association analysis of gene variants and diseases, as well as lifestyle habits and physical characteristic have attracted much attention. In our work we focus on the large-scale statistical association analysis, as seen in Genome Wide Association Study (GWAS) that targets the entire genome-wide scale data, and propose new methods to speed up the statistical analysis of disease-causal gene study on relational databases.

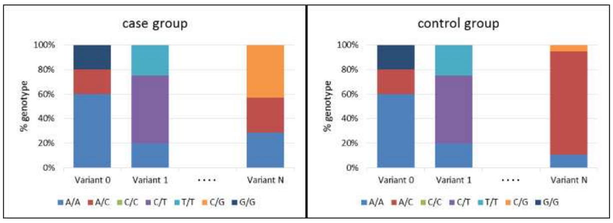


Figure 1: Example of genotype distributions

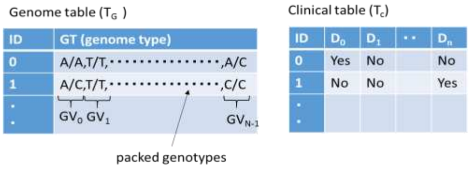
Disease-causal gene study aims at finding genotypes (types of gene variants) that are relevant to a target disease. This is done by dividing patients into a group with disease (case group) and another without disease (control group), and finding genotypes with different distributions between the two groups. For example, in Figure 1, there is no difference in genotype distribution between the case group and the control group for variants 0 and 1. However, since a significant difference can be seen in the distribution for variant N, this variant may be relevant to the target disease. The association between the target disease and the variants is calculated by performing a significance difference test under the null hypothesis (Cochrane-Armitage test, χ^2 test, etc.) for each gene variant, and when the p value falls below the significance level, the variant is considered relevant to the disease.

The processing required for the described association analysis is thus executed in two steps: (1) aggregating to generate the genotype distribution and (2) significant difference test. For the huge amount of genome data available nowadays step (1) requires much processing, and the total analysis may require some days if naively executed. Tools such as plink [3] address the statistical processing used in GWAS, and the performance of those analysis has shown improvements recently. In those tools, it is common to use genome information in a flat format file such as VCF, pad, or bad format [3]. However, in association analysis, the selection of the population greatly affects the correctness of the statistical result. Therefore, to search for meaningful results, an iterative approach where the selection of the population and the corresponding statistical processing are repeated a number of times is necessary to obtain correct statistical results. In order to repeatedly select and process data, it would be much easier if all the necessary data such as the genome data as well as the physical characteristic data (sex, race, age, etc.), the medical treatment data (presence/absence of disease, diagnosis results, etc.) related to the analysis were stored and managed in a single RDBMS not in flat format files. Recently, some studies that store and manage the whole necessary data in RDBMS have been proposed [4] [5], but none of the studies focus on high-speed processing of large-scale data on those RDBMS. It motivated us to speed up the processing on the data in RDBMS. In the previous work [1], we proposed a novel genome type and aggregate function as extended user-defined types of PostgreSQL and developed a new genotype data structure for efficient aggregate processing. However, in order to handle actual data, here we extend our previous work to handle variants with variable number of genotypes and to further speed-up the processing of large scale data.

In this paper, we first review our genome type and aggregate function in Section 2. Then, we introduce a new data structure to handle variants with variable size genotypes, and parallelization to handle large scale data in Sections 3 and 4, respectively. We finalize with some concluding remarks in Section 5.

2. Genome type and aggregate function

In our previous work [1], we have introduced a new genome type and aggregate function as extensions of PostgreSQL, and verified that those extensions enabled efficient genome analysis on RDBMS. Figure 2 illustrates genome table T_G where each row stores an individual ID and a field of genome type (GT) that packs the genotypes of all the N genome variants ($GV_0 \ldots GV_{N-1}$). Additional data on each individual such as clinical, demographic, lifestyle information, etc. can also be stored in other tables. Figure 2 illustrates a clinical table T_C that registers the disease of each individual. SQL 1 shows the SQL statement when executing aggregation using the genome type aggregate function fjgeno_count(). Since all the genotypes of the gene variants were packed in a single field, the genome aggregate function could efficiently count up through all genotypes of each individual at once.



Genome table (T_G)

ID	GT (genome type)
0	A/A,T/T,················,A/C
1	A/C,T/T,················,C/C
·	
·	

Clinical table (T_c)

ID	D_0	D_1	··	D_n
0	Yes	No		No
1	No	No		Yes
·				
·				

Figure 2: Database schema of genome type

```
SELECT fjgeno_count(TG.GT) FROM TG
WHERE TG.ID = Tc.ID AND Tc.D0 = YES;
```

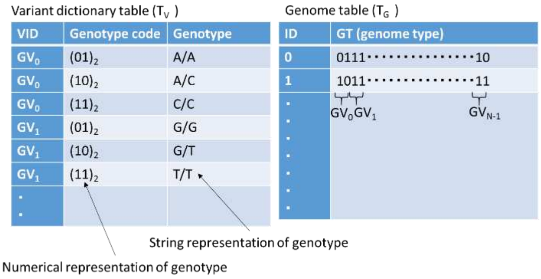
SQL 1: Statement with genome type aggregation function

3. New data structure for genome type

3.1 Dictionary-based encoding

3.1.1 Data structure

As illustrated in Figure 2, the genome type introduced in our previous work stored the genotype information in text format with all the genotypes as strings packed in one line and delimited by comma. In order to further improve its efficiency, here we introduce the utilization of a dictionary to compress data by numerically encoding the genotypes. Note that most gene variants can have a few variations of genotype patterns. For instance, a simple SNP can have three genotype patterns 'A/A', 'A/C' or 'C/C'. When represented in text format, a genotype 'A/A' plus a delimiter character would need four bytes. However, a numeric code requires only two bits and thus has a compression factor of 1/16.



Variant dictionary table (T_V)

VID	Genotype code	Genotype
GV_0	$(01)_2$	A/A
GV_0	$(10)_2$	A/C
GV_0	$(11)_2$	C/C
GV_1	$(01)_2$	G/G
GV_1	$(10)_2$	G/T
GV_1	$(11)_2$	T/T
·		
·		

Genome table (T_G)

ID	GT (genome type)
0	0111················10
1	1011················11
·	
·	
·	
·	
·	
·	
·	

GV_0GV_1
GV_{N-1}
String representation of genotype
Numerical representation of genotype

Figure 3: Two-bit encoded genome type and variant dictionary table

Figure 3 illustrates a variant dictionary table (T_V) that, for each of the N variants ($GV_0 \ldots GV_{N-1}$), maps the genotype to its numeric code. In this example, each GV_i can have three types of genotypes ('A/A', 'A/C', 'C/C' for GV_0, 'G/G', 'G/T', 'T/T' for GV_1, …) and thus each GV_i can be encoded using two bits ($(01)_2$, $(10)_2$, $(11)_2$). Therefore the genome table T_G is represented as an array of 2N bits to store the genotypes of the N genome variants.

3.1.2 Performance of the new data structure

The proposed dictionary-based data structure also contributes to the acceleration of the aggregate processing by avoiding heavy text parsing processing. We measured the execution time of the query in SQL 1 using the genome type based on dictionary-encoding and on text developed in our previous work. We executed the query on a PRIMERGY RX2540 M1 machine with Xeon E5-2660 3 @2.60GHz, 576GB memory, and shared buffers = 128GB for the PostgreSQL configuration parameter, using data of 100,000 individuals with 100,000 variants. As shown in Table 1, while the execution time of aggregate processing takes about 46 seconds when parsing text, it takes only about 8 seconds using the proposed dictionary method, which results in more than 5x faster execution.

Table 1: Execution time of aggregate processing

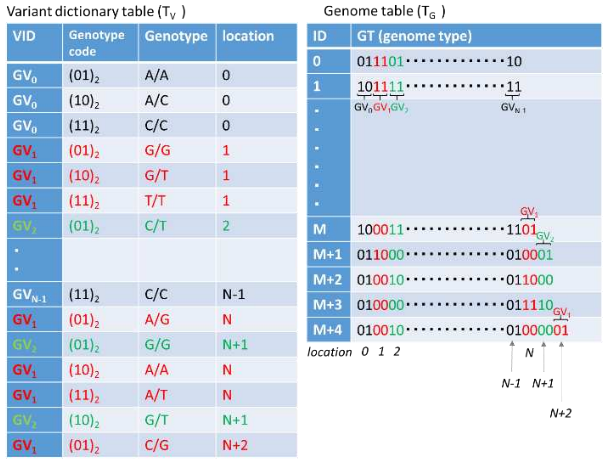
Genome Type	Execution time (sec)
Text parsing (Previous work)	45.743
Dictionary based (This work)	8.331

3.2 Support for variable genotypes

3.2.1 Dynamic data structure

As shown in 3.1, a dictionary encoding the genotypes to a fixed length bitwise code can compress data and improve the aggregate processing. As illustrated in Section 3.1, a large number of variants presents a small number of genotype patterns, namely three. However, even some simple SNPs and INDELs can present more than three patterns, and especially STRs (short tandem repeat) have many repeated patterns and so can present more than a hundred patterns. We can easily come up with two naïve methods to handle the variable number of genotype patterns. One way is to use a fixed length code that is long enough to represent the maximum number of possible genotype patterns among all variants. Another way is to use different length codes for each of the variants so that each variant code is long enough to represent the maximum number of possible patterns for that variant. However, note that both approaches are static and require a prior knowledge of the maximum number of patterns for the variants. In case a new individual's genome type is inserted with a new pattern that requires a longer variant code, both approaches require the reconstruction of the genome table as well as the dictionary. We believe that the preknowledge of the maximum number of genotype patterns is not practical in real genome analysis where new data emerges constantly. Since the reconstruction of dictionary and genome table are too heavy, we introduce a new dynamic method that efficiently handles new genotype patterns without prior knowledge.

Our dynamic approach handles new patterns that exceed the capability of initial bitwise codes by appending new bitwise codes at the end of our data structure. As illustrated in Figure 4, let's suppose that for the first M individuals, all the N genome variants contain only three patterns that are represented by two-bit codes $(01)_2$, $(10)_2$ and $(11)_2$. Therefore, for individuals 0 to M-1, GT is an array with two-bit spaces allocated to each of the N variants. Now, let's suppose that an individual M with a genome pattern 'A/G' for GV_1, that is different to the previous patterns 'G/G', 'G/T' and 'T/T' that appeared so far in GV_1, is inserted. In that case, the genome array for individual M is extended with a new two-bit space, and 'A/G' is encoded as $(01)_2$ for this new two-bit space for GV_1, while the original two-bit space for GV_1 is inserted with code $(00)_2$ indicating that the necessary code is stored elsewhere in a new two-bit space. Note that the dictionary T_v is updated and a new entry gives that 'A/G' is encoded as $(01)_2$ for GV_1 and the code is located at the Nth two-bit space in GT array.



Variant dictionary table (T_V)

VID	Genotype code	Genotype	location
GV_0	$(01)_2$	A/A	0
GV_0	$(10)_2$	A/C	0
GV_0	$(11)_2$	C/C	0
GV_1	$(01)_2$	G/G	1
GV_1	$(10)_2$	G/T	1
GV_1	$(11)_2$	T/T	1
GV_2	$(01)_2$	C/T	2
. .			
GV_{N-1}	$(11)_2$	C/C	N-1
GV_1	$(01)_2$	A/G	N
GV_2	$(01)_2$	G/G	N+1
GV_1	$(10)_2$	A/A	N
GV_1	$(11)_2$	A/T	N
GV_2	$(10)_2$	G/T	N+1
GV_1	$(01)_2$	C/G	N+2

Genome table (T_G)

ID	GT (genome type)
0	011101··············10
1	101111··············11
. . . . . .	
M	100011··············1101
M+1	011000··············010001
M+2	010010··············011000
M+3	010000··············011110
M+4	010010··············01000001

Figure 4: Adding genotype patterns

Analogously, the procedure continues as follows:

- ✓ When a fourth genotype pattern 'G/G' newly appears for GV_2 by inserting individual M+1, another space is added for GV_2 following the second space for GV_1 with code $(01)_2$, and the first space for GV_2 is filled with $(00)_2$;
- ✓ When a fifth pattern 'A/A' newly appears for GV_1 by inserting individual M+2, the code $(10)_2$ is stored in the existing second space for GV_1, and the first space for GV_1 is filled with $(00)_2$;
- ✓ When a sixth pattern 'A/T' for GV_1 and a fifth pattern 'G/T' for GV_2 appear by inserting individual M+3, they are stored in the existing second spaces for GV_1 and GV_2 as codes $(11)_2$ and $(10)_2$, and the other spaces for GV_1 and GV_2 are filled with $(00)_2$, respectively;
- ✓ When a seventh pattern 'C/G' newly appears for GV_1 by inserting individual M+4, a third space is needed for GV_1. In this case, the third space for GV_1 containing $(01)_2$ is added following the second space for GV_2, and $(00)_2$ is inserted into the first and second spaces for GV_1;

Note that all these information are registered on the variant dictionary table T_v that is extended with a location field that gives the two-bit space in which a pattern for each of the variants is found. The physical address of such location is decided when loading the genome data, and then when new patterns are inserted and new space is needed to store them.

In our method a fast array access is feasible because of its fixed length bitwise codes, and although the information for one variant is distributed among several spaces, each summation is efficiently processed over two-bit fixed elements. After the counts for each two-bit fixed element are processed, they are aggregated to generate the final counts for each variant by using the information on the variant dictionary table.

3.2.2 *Efficiency of the dynamic data structure*

To evaluate the efficiency of this method, we run the query on SQL 1 using the data distribution shown in Table 2. It emulates actual data containing 90,000 normal SNPs with three genotype patterns, 9900 irregular SNPs with six genotype patterns, and 100 STRs with 55 genotype patterns.

Table 2: Example of genotype size distribution

Counts	# of genotype pattern	Supposed variant type
90000	3	2-allelic SNPs
9900	6	3-allelic SNPs
100	55	STRs

Table 3 shows the processing times when all variants have only three patterns (Fixed), and for the case of Table 2 (Variable). We found that even for the case of variable variant patterns, the processing time has only a 20% increase over the "ideal" case of fixed three patterns, which is proportional to the increase in table size. Therefore, we found that the proposed dynamic approach is very efficient in handling genome data without any prior knowledge on the number of variant patterns.

Table 3: Times for fixed and variable patterns

	Fixed	Variable
Time (sec)	8.331	9.958
Size (GB)	2.70	3.32

4. Acceleration by parallelization

4.1 Query parallelization

With the aid of the increasing number of processors and cores in one machine, parallelization is a very effective way for processing speed acceleration. Here we show how the processing of our proposed genome data structures on PostgreSQL could take advantage of parallel processing. Since parallel query capabilities were introduced in the latest release 9.6 of PostgreSQL [2], we run the query SQL 1 on PostgreSQL 9.6. However, as a result of our trial, we found that the introduced parallelization is not effective for performance acceleration of that query. Figure 5 shows which processing are parallelized in the query execution plan created when running the query of SQL 1. We can see that PostgreSQL 9.6 parallelizes the join between

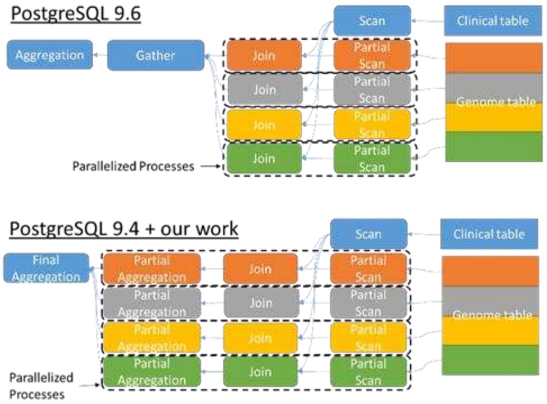


Figure 5: Parallelized query plan

the genome table and the clinical table, however the subsequent aggregate of the large join result is not parallelized and thus can become the bottleneck of the total query processing.

As we show later in Section 4.3, a more efficient parallel execution was achieved when using our original parallel function developed in a previous work [6] based on PostgreSQL 9.4. As shown in Figure 5, this function parallelizes the query execution plan from the topmost plan and thus both join and aggregation are parallelized, resulting in the acceleration of the total performance of the query.

4.2 SIMD processing

In addition, SIMD instructions of the processor could be used to parallelize the processing in the CPU instruction. Currently we can run 256-bit wide vector processing utilizing AVX2 which has been equipped from Sandy Bridge generation of Intel Xeon processors. The aggregate processing is mainly composed of two steps: taking the genotype patterns stored in each variant at the genome type structure, and then, counting the corresponding variant elements in a summation array. In order to efficiently realize the vector processing, we introduce a lookup array which stores vectors corresponding to the pattern values of the genome type and that should be added to the summation array. Figure 6 illustrates how the count-up processing uses the lookup array. First, the value $(00010110)_2$ composed of four variant's pattern codes is taken from the genome type. The decimal value of $(00010110)_2$ is 22 and thus, the 22th element of the lookup array gives the vector with the bits where places corresponding to the four variant's pattern values (GV_n's $(00)_2$, GV_{n+1}'s $(01)_2$, GV_{n+2}'s $(01)_2$, GV_{n+3}'s $(10)_2$) are set to 1. Then, one counting cycle is done by adding the vector to the summation array with SIMD instruction. Note that all the 256 vector values that corresponds to the composition of the four variant's pattern codes are prepared on the lookup array in advance.

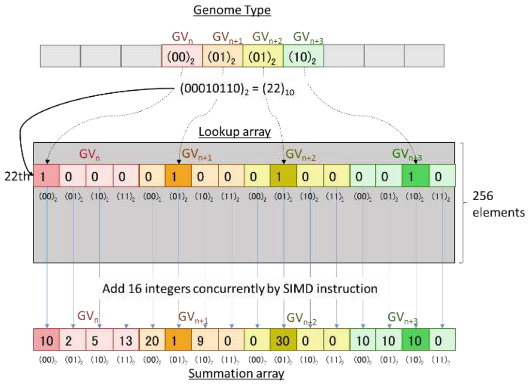


Figure 6: SIMD processing using lookup array

Because the element size of the summation array has to fit one SIMD vector, its size is restricted to 16-bit width. Thus when any element in the summation array achieves the maximum value of 65535 in integer, all the element values are added to a "final summation array" whose elements are wide enough. The summation array elements are then cleared to continue the aggregate processing.

4.3 Performance evaluation

In this section, we evaluate how the parallel processing of the newly proposed genome type accelerates the aggregate processing. We used the same experimental environment in Section 3.1.2. The results are shown in Figure 7 when using PostgreSQL 9.6 and 9.4 extended with our parallel function.

We can see that for one core, the newly released PostgreSQL 9.6 was improved from the older 9.4. However, as we explained in 4.1, since PostgreSQL 9.6 cannot parallelize the aggregate that represents the heaviest operation in the query, its total time does not scale when increasing the number of cores. On the other hand, for PostgreSQL 9.4 with our parallel extension, the execution time of 8.3 seconds on single core improves to 2.1 seconds on eight cores, i.e. about four times faster. In addition, using SIMD instructions, the performance improves by 20% compared with simple aggregate processing using dictionary encoded genome type.

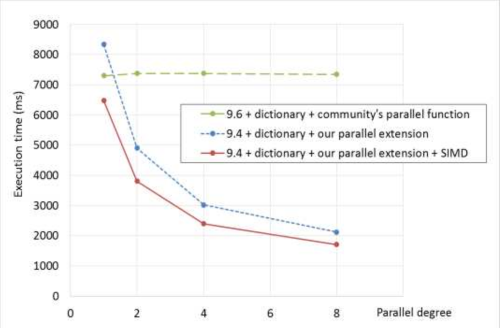


Figure 7: Execution time of aggregate processing

5. Conclusion

In this paper, we present our new efforts on accelerating genomic analysis to deal with actual large-scale data. The dictionary-based data structure described in 3.1.2 enables dealing with genotype patterns of variable maximum number and resulted in 5 times faster aggregate processing. And as shown in the results in 4.3, query parallelization with 8-cores and SIMD processing resulted in 5 times faster aggregate processing, and thus resulting in a total acceleration factor of 25x. This results in an execution time of less than two seconds for the aggregation of genome variants for genome information of 100 thousands individuals with 100 thousands variants. In recent studies, it is reported that more than 10 million variants have been included in the human genome. For such huge data, our aggregate processing would require some hundreds seconds. We believe that meaningful association studies requires a try and error approach where a variety of conditions are run iteratively. We believe our method could contribute to the feasibility of such an iterative approach for genome analysis.

6. REFERENCES

[1] Y. Ujibashi, M. Kawaba, L. Harada (2015) Proposal of Database Type and Aggregation Function for Accelerating Medical Genomics Study on DBMS EDBT 2016: **672-673**

[2] The PostgreSQL Global Development Group. (1996-2016) *PostgreSQL* http://www.postgresql.org/

[3] C. C Chang, C. C Chow, L. CAM Tellier, S. Vattikuti, S. Purcell, J. J Lee (2015) Second-generation PLINK: rising to the challenge of larger and richer datasets. GigaScience, **4**.

[4] A. Ameur, I. Bunkikis, S. Enroth, et al. (2014) CanvasDB: a local database infrastructure for analysis of targeted- and while genome resequencing projects. *Database*, Vol. 2014, Article ID bau098

[5] U. Paila, B. A. Chapman, R. Kirchner (2013) GEMINI: integrative exploration of genetic variation and genome annotations. *PLOS Comput. Biol.*,**9**,e1003153

[6] Y. Ujibashi, M. Nakamura, T. Tabaru., T. Hashida, M. Kawaba, L. Harada.: Design of a Shared Memory mechanism for efficient parallel processing in PostgreSQL. IISA 2015: 1-6

Authority-Based Team Discovery in Social Networks

Morteza Zihayat[*,$], Aijun An[$], Lukasz Golab[†], Mehdi Kargar[‡], Jaroslaw Szlichta[#]
[*]University of Toronto, Toronto, Canada, [$]York University, Toronto, Canada
[†]University of Waterloo, Waterloo, Canada, [‡]University of Windsor, Windsor, Canada
[#]University of Ontario Institute of Technology, Oshawa, Canada
mori.zihayatkermani@utoronto.ca, aan@cse.yorku.ca, lgolab@uwaterloo.ca,
mkargar@uwindsor.ca, jaroslaw.szlichta@uoit.ca

ABSTRACT

Given a social network of experts, we address the problem of discovering a team of experts that collectively hold a set of skills required to complete a given project. Prior work ranks possible solutions by communication cost, represented by edge weights in the expert network. Our contribution is to take experts' authority into account, represented by node weights. We formulate several problems that combine communication cost and authority, we prove that these problems are NP-hard, and we propose and experimentally evaluate greedy algorithms to solve them.

1. INTRODUCTION

An expert network is a social network containing professionals who provide specialized skills or services. Expert network providers include the employment-oriented service LinkedIn, the repository hosting service GitHub, and bibliography-based Websites such as DBLP and Google Scholar. A node in an expert network corresponds to a person and node labels denote his or her areas of expertise. Experts may be connected if they have previously worked together, co-authored a paper, etc. Edge weights may denote the strength of a relationship, the number of co-authored publications, or the *communication cost* between experts [4, 5].

There has been recent interest in the problem of finding teams of experts from such networks; see, e.g, [3, 5]. A common approach has been to find a subgraph of the expert network whose nodes collectively contain a given set of skills and whose communication cost is minimal. In this paper, we argue that in many practical applications, other factors should also be considered. For example, experts may be associated with *authority* metric such as h-index or number of publications. Here, we may want to minimize communication costs and maximize authority. Furthermore, in large social networks, experts holding the desired skills may not be directly connected. Thus, we may obtain a subgraph with some nodes, the *skill holders*, corresponding to team members who have the desired skills, and other nodes serving as *connectors*. The authority of connectors may also affect the quality of the team; e.g., connectors may serve as mentors for the skill holders.

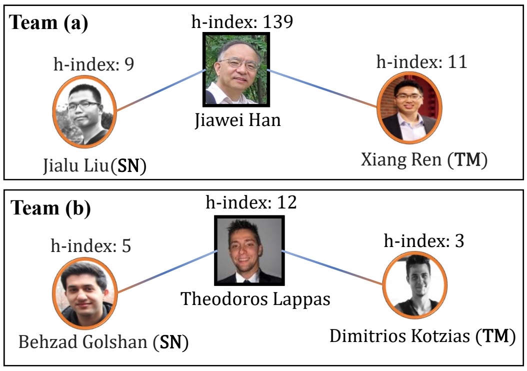


Figure 1: Two teams with expertise in SN and TM.

For instance, consider the two teams of researchers in Figure 1, both having expertise in social networks (*SN*) and text mining (*TM*). Team (a) and (b) both have two skill holders and a connector node; in this example, we use graduate students as skill holders and professors as connectors. Assuming equal communication costs, i.e., each edge having the same weight, previous work cannot distinguish between these two teams. However, the experts in team (a) have higher authority (h-index). Furthermore, even if all the skill holders were to have the same authority, team (a) may be preferable because its connector has higher authority.

Our contributions are as follows.

1. We formally define the problem of authority-based team formation in expert networks. We formulate three ranking objectives which optimize communication cost, skill holder authority, connector authority and combinations of them. We prove that optimizing these objectives is NP-hard.
2. Since these problems are NP-hard, we propose greedy algorithms to solve them. We present an algorithm to optimize communication cost over an expert network G. We then give a transformation which moves authority (node weights) onto the edges of a new graph, G', and prove that our algorithm also optimizes the other objectives over G'.
3. We perform a comprehensive evaluation using the DBLP dataset to confirm the effectiveness and efficiency of our approach. In particular, we show that the teams discovered by our techniques perform higher-quality research than those found using prior work.

2. PRELIMINARIES

Let $C = \{c_1, c_2, \ldots, c_m\}$ be a set of m experts, and $S = \{s_1, s_2, \ldots, s_r\}$ be a set of r skills. An expert c_i has a set of skills, denoted as $S(c_i)$, and $S(c_i) \subseteq S$. If $s_j \in S(c_i)$, expert c_i has skill s_j. Furthermore, a subset of experts $C' \subseteq C$ have skill s_j if at least one of them has s_j. For each skill s_j, the set of all experts having skill s_j is denoted as $C(s_j) = \{c_i | s_j \in S(c_i)\}$. A project $P \subseteq S$

Series ISSN: 2367-2005

10.5441/002/edbt.2017.54

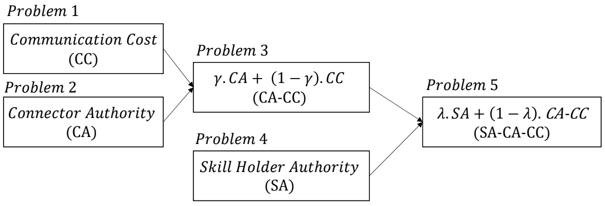


Figure 2: The Proposed Approach

is a set of required skills. A subset of experts $C' \subseteq C$ *covers* a project P if $\forall s_j \in P\ \exists\ c_i \in C', s_j \in S(c_i)$.

We model the social network of experts as an undirected graph G. Each node in G is an expert in C (we use the terms expert and node interchangeably). Each expert c_i has an application-dependent authority $a(c_i)$. To convert authority maximization into a minimization problem, we set $a'(c_i) = \frac{1}{a(c_i)}$. Furthermore, let $w(c_i, c_j)$ be the weight of the edge between two experts c_i and c_j. Edge weights correspond to application-dependent communication cost or relationship strength. There is no edge between experts who have no relationship or prior collaboration. Formally:

Definition 1. **Team of Experts:** Given an expert network G and a project P that requires the set of skills $\{s_1, s_2, \ldots, s_n\}$, a *team of experts* T is a connected subgraph of G whose nodes cover P. With each team, we associate a set of n skill-expert pairs: $\{\langle s_1, c_{s_1}\rangle, \langle s_2, c_{s_2}\rangle, \ldots, \langle s_n, c_{s_n}\rangle\}$, where c_{s_j} is an expert in T that has skill s_j for $j = 1, \ldots, n$.

The same expert may cover more than one required skill, i.e., c_{s_i} can be the same as c_{s_j} for $i \neq j$. Also, there may not be a direct edge between some two experts c_{s_i} and c_{s_j} in G. Thus, T may include connector nodes that may not hold any skill in P (e.g., Han and Lappas in Figure 1). Assuming that edge weights denote communication costs, minimizing communication costs amounts to minimizing the sum of the weights of the team's edges [3].

Definition 2. **Communication Cost (CC):** Suppose the edges of a team T are denoted as $\{e_1, e_2, \ldots, e_t\}$. The communication cost of T is defined as CC(T) $= \sum_{i=1}^{t} w(e_i)$, where $w(e_i)$ is the weight of edge e_i.

Problem 1. Given a graph G and a project P, find a team of experts T for P with minimal communication cost CC(T).

This is an NP-hard problem [3] which has been studied before. Extensions of this problem have also been considered, e.g., optimizing personnel cost and proficiency of skill holders [2, 7], or recommending replacements when a team member becomes unavailable [4]. However, to the best of our knowledge, existing approaches do not optimize the authority of skill holders and connectors.

3. TEAM FORMATION FRAMEWORK

3.1 Foundations

We are interested in optimizing both communication cost and authority. Note that we optimize the authority of connectors and skill holders separately. Some applications may find the authority of skill holders more important than that of the connectors (and vice versa), e.g., those where skill holders execute the project and connectors only provide guidance. Therefore, we optimize them with different tradeoff parameters, γ and λ, with respect to the communication cost and to each other. Figure 2 summarizes the problems we tackle and the remainder of this section discusses them in detail. First, we define the *connector authority* of a team as the sum of the inverse-authorities $a'(c_i)$ of its connectors.

Definition 3. **Connector Authority (CA):** Suppose that the connectors of a team T (all nodes excluding skill holders) are denoted as $\{c_1, c_2, \ldots, c_q\}$. The connector authority of T is defined as CA(T) $= \sum_{i=1}^{q} a'(c_i)$.

Problem 2. Given a graph G and a project P, find a team of experts T for P with minimal connector authority CA(T).

THEOREM 1. *Problem 2 is NP-hard.*

Due to space limitations, we refer the reader to the extended version of this paper (technical report) for all proofs [6]. Furthermore, we are interested in the bi-criteria optimization problem of minimizing CC and CA. To do so, we combine these two objectives into one with a tradeoff parameter γ (after normalizing edge and node weights since they may have different scales).

Definition 4. **CA-CC Objective:** Given a team T and a tradeoff parameter γ, where $0 \leq \gamma \leq 1$, the CA-CC score of T is defined as CA-CC(T) $= \gamma \times$CA(T) $+ (1-\gamma)\times$CC(T).

Problem 3. Given a graph G, a project P, and a tradeoff parameter γ, find a team of experts T for P with minimal CA-CC(T).

THEOREM 2. *Problem 3 is NP-hard.*

We are also interested in optimizing the authority of skill holders.

Definition 5. **Skill Holder Authority (SA):** Suppose that the skill holders of a team T are denoted as $\{c_1, c_2, \ldots, c_n\}$. The skill holder authority of T is defined as $SA(T) = \sum_{i=1}^{n} a'(c_i)$.

Problem 4. Given a graph G and a project P, find a team of experts T for P with minimal skill holder authority SA(T).

Problem 4 can be solved in polynomial time: for each skill in P, we find an expert with the highest a (lowest a'), and then produce a connected subgraph containing the selected experts. However, this ignores communication cost and connectors' authority. We now put all three objectives together.

Definition 6. **SA-CA-CC Objective:** Given a team T and a tradeoff parameter λ, where $0 \leq \lambda \leq 1$, the SA-CA-CC objective of T is defined as SA-CA-CC(T) $= \lambda \times$SA(T) $+ (1-\lambda)\times$CA-CC(T).

Problem 5. Given a graph G, a project P, and a tradeoff parameter λ, find a team of experts T for P with minimal SA-CA-CC(T).

THEOREM 3. *Problem 5 is NP-hard.*

Since the tradeoff parameters γ and λ are application-dependent, we leverage user and domain expert feedback to set and update them over time (see experiment in Figure 5). Incorporating user feedback is important for achieving high precision.

3.2 Search Algorithms

Since Problems 1, 2, 3 and 5 are NP-hard, we propose efficient and effective greedy algorithms to solve them in polynomial time.

Optimizing CC: Algorithm 1 returns a subtree of G corresponding to a team with optimized communication cost (sum of edge weights). The for-loop in line 3 considers each expert c_r as a potential root node for the subtree (c_r may end up being a skill holder or a connector). To build a tree around c_r, for each required skill s_i, we select the nearest skill holder, denoted $bestExpert$, that contains s_i (lines 9-13; assume DIST(v_1,v_2) finds the shortest path, i.e., the smallest sum of edge weights, between two nodes v_1, v_2). The method add in line 13 connects the $bestExpert$ to the current team, meaning that any additional nodes along the path from the root to $bestExpert$ are also added. The tree with the lowest sum of edge weights is the best team (lines 14-17). To find the shortest path between any two nodes in constant time, we use *distance labeling*, or *2-hop cover* [1]. As a result, the complexity of

Algorithm 1 Finding Best Team of Experts

```
Input: graph $G$ with $N$ nodes; project $P = \{s_1, s_2, \ldots, s_t\}$; the set of experts that contains each skill $s_i$, $C(s_i)$, for $i = 1, \ldots, t$.
Output: best team of experts
 1: $leastTeamCost \leftarrow \infty$
 2: $bestTeam \leftarrow \emptyset$
 3: for $r \leftarrow 1$ to $N$ do
 4:     $root \leftarrow c_r$
 5:     $teamCost \leftarrow 0$
 6:     $team \leftarrow \emptyset$
 7:     set the root of $team$ to $root$
 8:     for $i \leftarrow 1$ to $t$ do
 9:         $minCost_i \leftarrow \min_{v \in C(s_i)} DIST(root, v)$
10:         $bestExpert \leftarrow \arg\min_{v \in C(s_i)} DIST(root, v)$
11:         if $bestExpert \neq \emptyset$ then
12:             $teamCost \leftarrow teamCost + minCost_i$
13:             $team$.add($bestExpert$)
14:     if size($team$) = $t$ then
15:         if $teamCost < leastTeamCost$ then
16:             $leastTeamCost \leftarrow teamCost$
17:             $bestTeam \leftarrow team$
18: return $bestTeam$
```

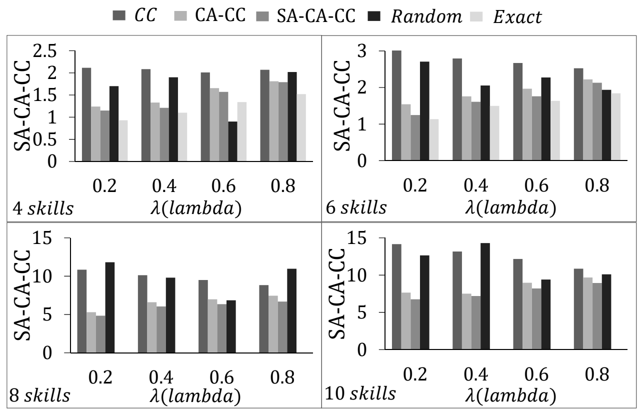


Figure 3: *SA-CA-CC* scores of different ranking methods(γ = 0.6)

Algorithm 1 is $O(N \times t \times |C_{max}|)$, where $|C_{max}|$ is the maximum size of the expert sets $C(s_i)$ for $1 \leq i \leq t$. The N comes from the for-loop in line 3, the t comes from the for-loop in line 8 and the $|C_{max}|$ is due to computing the shortest path to each expert in $C(s_i)$ in lines 9 and 10. For finding top-k teams, we initialize a list L of size k for the output. The list L is updated after each iteration of the loop and the new team is added to L if its cost is smaller than the last team in L. The runtime complexity remains the same as the entire operation only needs an extra pass over L in each iteration.

To solve the other problems, we transform the expert network G by moving authority (node weights) onto the edge weights and then running Algorithm 1 on the transformed graph.

Optimizing CA-CC: For Problem 3, we transform G into G' as follows. Let the edge weight between nodes c_i and c_j in G be $w(c_i, c_j)$. In G', we transform each edge weight to $w'(c_i, c_j) = \gamma(a'(c_i) + a'(c_j)) + 2 \times (1 - \gamma)w(c_i, c_j)$. The DIST function now finds shortest paths by adding up the *transformed* edge weights w'. However, we only want to take connector authority into account, not skill-holder authority. Therefore, in lines 9 and 10, we replace $DIST(root, v)$ by $DIST(root, v) - \gamma a'(v)$; note that v is always a skill holder. If $root$ contains skill s_i, then $DIST$ is set to zero and skill s_i is assigned to $root$. With this modification, we claim that running Algorithm 1 on G' optimizes CA-CC. Note that setting $\gamma = 1$ solves Problem 2, i.e., optimizes CA.

Optimizing SA-CA-CC: Recall that SA-CA-CC is a linear combination of communication cost, skill holder authority and connector authority. We re-use G' from above to capture communication cost and connector authority. Additionally, we need to take λ into account and add the contribution of skill holder authority. To do this, we replace $DIST(root, v)$ in lines 9 and 10 with $(1 - \lambda)(DIST(root, v) - \gamma a'(v)) + \lambda a'(v)$. Note that we have to subtract the authority of skill holders with parameter γ and then add it with parameter λ. As before, if $root$ contains skill s_i, then $DIST$ is set to zero and skill s_i is assigned to $root$. We claim that running Algorithm 1 with this modification, along with using G' instead of G, solves Problem 5.

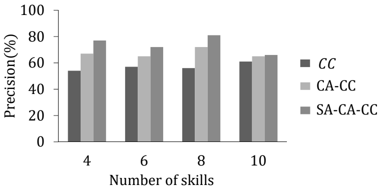


Figure 4: Precision of top-5 teams for different methods

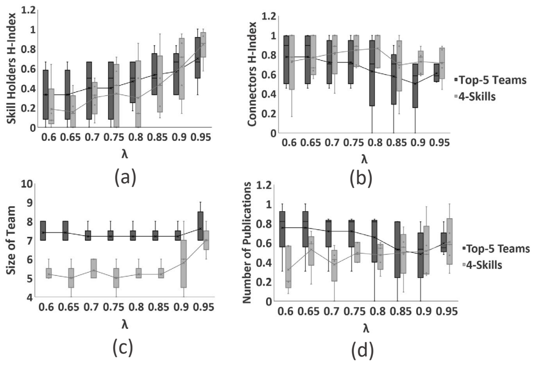


Figure 5: Sensitivity of normalized results to λ

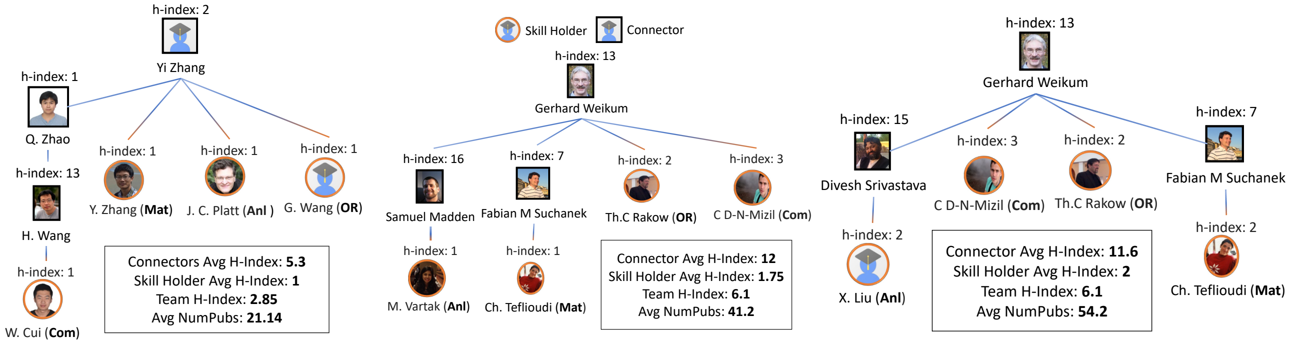
4. EXPERIMENTAL RESULTS

In this section, we use Algorithm 1 and its various modifications explained above to implement ranking strategies for team discovery which optimize CC, *CA-CC* and *SA-CA-CC*. CC corresponds to prior state-of-the-art, and our main goal is to show that *CA-CC* and *SA-CA-CC* are more effective. We also implemented $Random$, which randomly builds 10,000 teams and selects the one with the lowest *SA-CA-CC*, and $Exact$ which performs exhaustive search to find an (*SA-CA-CC*)-optimal solution. Note, however, that $Exact$ is intractable for large networks or large projects (containing many required skills). The algorithms are implemented in Java and the experiments are conducted on an Intel(R) Core(TM) i7 2.80 GHz computer with 4 GB of RAM.

Similar to previous work, we use the DBLP XML dataset[1] to build an expert graph [2, 3]. For potential skill holders, we take junior researchers with fewer than 10 papers and we label them with terms that occur in at least two of their paper titles. This gives us the areas of expertise. Similar to [2, 3], we set edge weights between two experts c_i and c_j to $1 - |\frac{b_{c_i} \cap b_{c_j}}{b_{c_i} \cup b_{c_j}}|$ (Jaccard Similarity) where b_{c_i} is the set of papers of author c_i. We use h-index as the node weight to denote authority. The resulting graph has 40K nodes (experts) and 125K edges. The number of skills in a project is set to 4, 6, 8 or 10. For each number of skills, we generate 50 sets of skills, corresponding to 50 projects, and we report average results over these 50 projects.

Exp-1 Effectiveness. We begin by comparing our *SA-CA-CC* ranking strategy with $Exact$; for completeness, we also test CC,

[1] http://dblp.uni-trier.de/xml/



(a) (CC)

(b) (CA-CC)

(c) (SA-CA-CC)

Figure 6: Best team of CC, *CA-CC* and *SA-CA-CC* with "skills": *analytics(***Anl***), matrix (***Mat***), communities(***Com***), object oriented(***OR***)*

CA-CC and $Random$, and compute their *SA-CA-CC* scores. Figure 3 plots the *SA-CA-CC* scores of different ranking strategies for different numbers of skills and different values of λ. For brevity, we fix γ at 0.6 but different values led to similar conclusions. We conclude that *SA-CA-CC* produces results that are close to those of $Exact$ (but note that $Exact$ was only able to handle 4 and 6 skills and did not terminate in reasonable time for 8 and 10 skills). Not surprisingly, *SA-CA-CC* has lower *SA-CA-CC* score than CC and *CA-CC*. We also note CC, *CA-CC* and *SA-CA-CC* have similar runtime since they use the same fundamental algorithm and indexing methods. The runtime depends on the number of required skills and is around a few hundred milliseconds (i.e., less than one second) on average.

Exp-2 User Study. We conduct a user study to evaluate the top-k precision of different ranking strategies. First, we create four projects with different numbers of required skills. Then, for each project, we run CC, *CA-CC* and *SA-CA-CC* and take the top-5 best teams returned by each. We give these results to six Computer Science graduate students, along with the average number of publications and the h-index of each expert included in the teams. We asked the students to judge the quality of the top-5 teams using a score between zero and one. Figure 4 shows the top-5 precision of each method. In this experiment, we set both λ and γ to 0.6. Both of our methods, *CA-CC* and *SA-CA-CC*, obtain better precision than CC for all tested projects.

Exp-3 Quality of Teams. We check if the top-5 teams returned by *CC* and *SA-CA-CC* were successful in real life. To do so, we examined the rankings of the publication venues of these teams according to the *Microsoft Academic* conference ranking. Since we used the *DBLP* dataset up to 2015 for team discovery, we only consider papers published in 2016. We set γ and λ to 0.6 and generate 5 different projects with four different skills. From the teams that co-authored papers in 2016, we found that 78% of the time the teams found by *SA-CA-CC* published in more highly-rated venues than those found by CC.

Exp-4 Sensitivity. Figure 5 shows the sensitivity of the results to λ (the tradeoff parameter between skill holder authority and *CA-CC*), specifically the sensitivity of the average h-index of skill holders (part a), the average h-index of connector nodes (part b), the average team size (part c) and the average number of publications (part d). Our methodology for evaluating sensitivity is as follows. First, we examine the effect of λ on the top 5 teams returned by *SA-CA-CC*. Given the project [*analytics, matrix, communities, object oriented*], *SA-CA-CC* finds top-5 teams using different values of λ. Second, we evaluate the effect of λ on a best team returned by *SA-CA-CC* for m different projects. For this, we randomly generate five projects with four skills each. Then, for each value of λ, *SA-CA-CC* finds the best team for each project. As shown in Figure 5, the measures change slowly as λ increases. We also observe that changing the value of λ by less than 0.05 does not affect the results and the quality of the team remains the same.

Exp-5 Qualitative Evaluation. Figure 6 illustrates the teams returned by CC, *CA-CC* and *SA-CA-CC* for the project [*analytics, matrix, communities, object oriented*]. Observe that CC returns a team with lower authority (average h-index) and average number of publications than *CA-CC* and *SA-CA-CC*. Moreover, Figure 6 shows that the skill holders of the team returned by *CA-CC* and *SA-CA-CC* are connected through authors with a higher h-index, and thus have a higher referral authority. We argue that the teams returned by our algorithms are more effective than the one returned by CC since it reveals a deeper connection among the experts that may not have been discovered by existing team formation methods. Note that connectors may not be directly involved in performing a task, but may provide guidelines and support to skill holders.

5. CONCLUSIONS

In this paper, we studied the problem of team discovery from networks of experts. We formulated new ranking objectives that take communication costs among experts as well as expert authority into account. We proved that satisfying these new objectives is NP-hard and proposed heuristic algorithms. We demonstrated the effectiveness of our techniques on the DBLP dataset. Another way to jointly optimize the communication cost and expert authority objectives is to find a set of Pareto-optimal teams. In the future, we plan to develop algorithms to find such teams and rank them based on relevant measures of interestingness.

6. REFERENCES

[1] T. Akiba, Y. Iwata, and Y. Yoshida. Fast Exact Shortest-path Distance Queries on Large Networks by Pruned Landmark Labeling. In *SIGMOD*, pages 349–360, 2013.

[2] M. Kargar, M. Zihayat, and A. An. Finding Affordable and Collaborative Teams from a Network of Experts. In *SDM*, pages 587–595, 2013.

[3] T. Lappas, L. Liu, and E. Terzi. Finding a Team of Experts in Social Networks. In *KDD*, pages 467–476, 2009.

[4] L. Li, H. Tong, N. Cao, K. Ehrlich, Y. Lin, and N. Buchler. Replacing the Irreplaceable: Fast Algorithms for Team Member Recommendation. In *WWW*, pages 636–646, 2015.

[5] S. B. Roy, L. Lakshmanan, and R. Liu. From Group Recommendations to Group Formation. In *SIGMOD*, pages 1603–1616, 2015.

[6] M. Zihayat, A. An, L. Golab, M. Kargar, and J. Szlichta. Authority-based Team Discovery in Social Networks. *CoRR abs/1611.02992, Technical Report available at https://arxiv.org/abs/1611.02992, 6 pages*, 2016.

[7] M. Zihayat, M. Kargar, and A. An. Two-Phase Pareto Set Discovery for Three-objective Team Formation. In *WI*, pages 304–311, 2014.

Correlation-Aware Distance Measures for Data Series

Katsiaryna Mirylenka[*]
IBM Research - Zurich
kmi@zurich.ibm.com

Michele Dallachiesa
Skysense
michele@skysense.co

Themis Palpanas
Paris Descartes University
themis@mi.parisdescartes.fr

ABSTRACT

The field of data series processing has attracted lots of attention thanks to the increased availability of unprecedented amounts of sequential data. These data are then processed and analyzed using a large variety of techniques, most of which are based on the computation of some distance function. In this study, we evaluate the benefits of incorporating into the distance functions correlation measures, which enable us to capture the associations among neighboring values in the sequence. We propose three such measures, inspired by statistical and probabilistic approaches. We analytically and experimentally demonstrate the benefits of the new measures using the 1NN classification task, and discuss the lessons learned.

1. INTRODUCTION

The field of data series processing has seen a tremendous progress in the database community thanks to the increased availability of an unprecedented amount of data [17, 3, 16, 18, 13]. Any data series complex analysis task can be reduced to modeling a distance measure that captures the most discriminating features across different classes or patterns in the data [12].

The most widely used distance models are variations of the Euclidean distance and are characterized by the invariant properties that they support. For example, the Dynamic Time Warping (DTW) distance [1] allows accelerations and decelerations of the signal along the x-axis, and the Longest Common Subsequence (LCSS) distance [7] allows gaps in the sequence. The Euclidean distance is widely used, and has been shown to be very effective for large data collections, performing equally well or outperforming new distance models (such as SpADe and TQuEST), as well as traditional elastic distance measures (such as DTW)[9]. Therefore, in this work we will concentrate on Euclidean distance.

We observe that the distance measures mentioned above do not model the correlations that do exist among neighboring points in the series. Nevertheless, previous work has shown that modeling explicitly the correlation inherent in the data series leads to better results [4, 5, 6]. An example is illustrated in Figure 1. The graph shows four series, namely X, Y, Z and Q. The point values of the series are the following: $X =< 2, 3, 2, 3, 2 >$, $Y =< 2, -1, 2, -1, 2 >$, $Z =< -1, -2, -1, -2, -1 >$ and $Q =< 1, 1, 1, 1 >$. The Euclidean distance between Q and the other series X, Y and Z is the same, $\sqrt{11}$. The series X and Z are equally similar to the series Q. Despite the larger deviations in the values of series Y, the distance between Q and Y is exactly the same. A similar result can be obtained for other Minkowski distances and their extensions, such as the DTW and LCSS distances, as well as for z-normalized series.

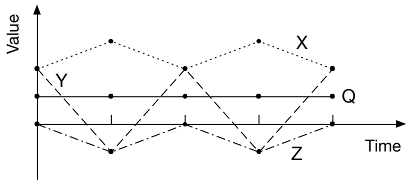


Figure 1: Euclidean distance fails to distinguish between the X and Y series, given query Q.

In this study, we answer the following question: can distance measures that take into account the neighboring-point correlations in the series outperform the Euclidean distance in mining tasks such as classification? As we will see, the answer to this question is *yes.*

In this work, we make the following contributions. We present distance models inspired by statistical and probabilistic approaches that have been designed to capture the correlation among neighboring points in a data series: autocorrelation, Markov chains and value-difference histograms defined over sliding windows. We combine the proposed models with the Euclidean distance and provide an experimental evaluation with real datasets, which demonstrates the utility of the correlation-aware distance measures.

2. NEED FOR A NEW DISTANCE

A data series X is a sequence of real valued points $X = \{x_i\}_{i=1}^n$ where n is the length of X, and x_i is the value of data series X at position i. A data series is *z-normalized* (or simply *normalized*)if its mean is equal zero and its variance is equal to one. The Euclidean distance between data series X

[*] Work done while at the University of Trento, Italy.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.55

and Y is defined as follows: $D_{Eucl}(X,Y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$. Though it is very efficient in many applications, euclidean and euclidean-like distances cannot capture correlations among neighboring data points in the sequence.

We note that the Euclidean distance between two series X and Y, formally denoted by $D_{Eucl}(X,Y)$, is invariant to two transform rules as defined below. First, a pair of corresponding points x_i and y_i can be swapped with any other pair of points x_j and y_j, $i \neq j$ without any change in the distance value. For example, the Euclidean distance between series $X =< 1,2,3,4 >$ and $Q =< 5,6,7,8 >$ does not change if we swap the second and the fourth values (obtaining series $X' =< 1,4,3,2 >$ and $Q' =< 5,8,7,6 >$, respectively), though the new series are obviously not the same.

Second, the value of the Euclidean distance does not change when new values x'_i and x'_j are assigned respectively to points x_i and x_j, where x'_i and x'_j satisfy the following condition:

$$(x_i - y_i)^2 + (x_j - y_j)^2 = (x'_i - y_i)^2 + (x'_j - y_j)^2$$

Consider for instance, the Euclidean distance between series $Q = \{0,0,0,0\}$ and $X = \{5,5,5,5\}$ is the same to the Euclidean distance between the series Q and $Y = \{4.3563, 5.5698, 4.3563, 5.5698\}$. The Euclidean norm distance for both pairs Q, X and Q, Y is 10, while the shape of the series is drastically different.

We conclude that Euclidean distance fails to capture important semantics of data series, as shown in the above examples. In contrast, the correlation-aware distance measures presented in Section 3 aim to reveal such differences.

3. PROPOSED DISTANCE MEASURES

In this section we introduce and describe four distance measures which take into account the correlations among neighboring points in the series.

3.1 Autocorrelation Distance (ACD)

The distance measure based on autocorrelation coefficient is not new, it comes from the statistical domain and is widely exploited in a data mining community [10]. In this work, we calculate the autocorrelation vector $R = \{r(\tau)\}_{\tau=1}^{n}$, which consists of autocorrelation coefficients $r(\tau)$ with different lags up to n: $r(\tau) = \frac{E[(x_t-\mu)(x_{t+\tau}-\mu)]}{\sigma^2}$, μ is a mean and σ^2 is a variance of a data series $X = x_i$. The distance between two series is defined as the Euclidean distance between their autocorrelation vectors. The length of aoutocorrelation vector n is a training parameter.

3.2 Markovian Distance

Markovian models are commonly used to capture correlations among points of a data series. A Markov chain of order k is a sequence of random variables, which satisfy the Markovian property that the current state of the chain depends only on the previous k states. In our study, we consider Markov chains with alphabet size $m = 32$, and treat the order as a parameter, which we need to estimate during the training phase. For the testing phase, we estimate a transition probability matrix M, which characterizes a Markov chain by estimating the conditional probabilities of the query X. We do this by looking across the series and first calculating the frequencies of all sequences of length k and $k+1$, and then calculating all the conditional probabilities: $M(x_{t-k}, x_{t-k+1}..., x_t) = Pr[x_t|x_{t-1},...,x_{t-k}] = \frac{Freq[x_t,x_{t-1},...,x_{t-k}]}{Freq[x_{t-1},...,x_{t-k}]}$), where $t = k+1,...,n$, n is the length of the series. We then identify the nearest neighbor, that is, the series Y with the highest probability of being generated by the model of the query series:

$$Pr(y_1,...,y_n|M) = Pr[y_1,...,y_k] \prod_{t=k+1}^{n} M(y_{t-k},...,y_t), \quad (1)$$

where $Pr[y_1,...,y_k]$ is the initial state of the Markov chain. In order to avoid the accumulation of machine error caused by the multiplications in Equation 1, we calculate the log of the probabilities:

$$\log Pr(y_1,.....,y_n|M) \sim \sum_{t=k+1}^{n} \log[M(y_{t-k}, y_{t-k+1}..., y_t)]. \quad (2)$$

This leads to a natural distance measure, which is a probability that one sequence is generated using a model of another sequence. As $\log Pr$ defined by Equation 2 is a similarity measure, the distance between X and Y can be defined as $-\log Pr$. Note that this distance can also be efficiently computed in an online setting, where streaming series for very large alphabet sizes should be compared, using Conditional Heavy Hitters [15, 14] for estimating the most significant elements of the transition probability matrix.

3.3 Local Distance Distribution (LDD)

In this section, we propose the Local Distance Distribution (LDD), a ranking function that is based on the distribution of Euclidean distances determined on sub-sequences from candidate series X_i and query Q.

Given a series X_i, let $X_i^{[a,b]}$ be the sub-sequence of X_i between positions a and b. Let $W_h(X_i, w)$ be the content of the sliding window on series X_i of length w whose first point is x_h, i.e., $W_h(X_i, w) =< x_h, ..., x_{h+w-1} >$. The set of distance samples between X_i and Q is denoted by $D(Q, X_i)$ and is defined as: $D(Q, X_i) = \{Euclidean(W_h(X_i, w), W_h(Q, w)) : h \in \{1, ..., n-w+1\}\}$, where $Euclidean(X_i, X_j)$ denotes the Euclidean distance between series X_i and X_j and n is the length of the series. $D(Q, X_i)$ is a set of pairwise point distances along the series Q and X_i. Let H_i be the equi-width histogram composed of B buckets that summarizes the distance values in $D(Q, X_i)$.

Given two series X_i and X_j, the probability that a random distance value $d_i \in D(Q, X_i)$ is lower than a random distance value $d_j \in D(Q, X_j)$ can be estimated as follows: $Pr(d_i < d_j) = \sum_{b=1}^{B} H_{i,b} \sum_{l=b+1}^{b} H_{i,l}$, where $H_{i,l}$ is the value of the lth bucket of the equi-width histogram H_i. We can now introduce the probability for a candidate series X_i to be the nearest neighbor to a query series Q as:

$$PNN(X_i, Q) = \prod_{j \neq i} Pr(d_i < d_j), \quad (3)$$

where d_i and d_j are two random distance values from $D(Q, X_i)$ and $D(Q, X_j)$, respectively. The function $PNN(X_i, Q)$ is a ranking function that can be used to implement a nearest neighbor classifier.

3.4 Using the Proposed Methods

Using the Euclidean distance for 1NN classification leads to the fastest and simplest classification. In this work, we combine Euclidean distance with the proposed techniques for 1NN classification: when the discrimination confidence of the Euclidean distance is low, then we switch to using our techniques. In this way, we aim to combine the speed of Euclidean with the accuracy of the proposed techniques.

Given an oracle, we can choose to use our techniques only when Euclidean fails. In practice though, we have to predict when this will happen. We use the following strategy for this classification failure prediction [8]. First, we compute a confidence value based on the distances to the two nearest neighbors belonging to two different classes: $Conf = 1 - \frac{d_i}{min_{i \neq j} d_j}, d_j = \min\{dist(Q, X_j) | j \in C\}$. Then, we use the proposed distance measures when this confidence value is below some threshold. Our experiments show that the accuracy of this prediction is slightly above 75%, and fairly robust for thresholds between 0.2-0.8.

4. EXPERIMENTAL EVALUATION

We compare our methods to the simple and widely-used Euclidean distance for the 1NN classification task. We report the F1 measure: $F1 = 2 * \frac{precision * recall}{precision + recall}$, with $precision = \frac{tp}{tp+fp}$ and $recall = \frac{tp}{tp+fn}$, where tp, fp and fn represent true positives, false positives, and false negatives, respectively. Precision and recall are calculated for each class separately, and their arithmetic mean is used to calculate the mean F1 value.

We use 43 UCR datasets with normalized series of different lengths from several domains [11].

4.1 Results

In the first set of experiments, we perform a sanity check by comparing the accuracy of using the proposed distance measures in a 1NN classifier, against the accuracy of a random classifier. The results, depicted in Figure 2, show that all three methods consistently outperform the random classifier (i.e., points above the diagonal). This is especially true for the case where (with the help of an oracle) we use the three proposed methods only when Euclidean distance fails to identify the correct class (i.e., square green points).

We now focus on the performance of the ACD distance, shown in Figure 3. As mentioned in Section 3.1, the autocorrelation function is a cross-correlation of a data series with itself within a given time lag. The resulting autocorrelation vectors are then used to compute the Euclidean distance between the series. Figure 3(a) shows that the ACD distance assisted by failure-prediction performs better than Euclidean only for some of the datasets (i.e., points above the diagonal). Failure-prediction is used in the way described in Section 3.4, where we predict (with a less than perfect accuracy) the cases that the Euclidean-based classification fails. A close look at the experimental results reveals that ACD significantly improves the classification accuracy for several datasets. One such dataset is Trace, for which the classification accuracy with ACD is 100%, while the Euclidean distance based classification has an accuracy of only 76%.

Figure 3(b) shows that switching to ACD when we know for sure that the Euclidean distance will fail leads to a remarkable improvement in accuracy. Thus, using a perfect oracle for predicting failure of Euclidean distance based classification and then switching to ACD based classification shows significant accuracy improvement for all 43 datasets.

Classification based on the proposed Markovian distance uses the transition probability matrix for each query data series in order to capture the correlation among adjacent points in the sequence. This transition probability matrix is used to find the series of the training set, which is the most likely to be generated by the query model. Since estimating the Markov model requires data series with discrete values, we used iSAX2.0 [2] to generate 32 discrete states for our data series. The experiments focus on the effect of the order of the Markov chain on classification accuracy. Our cross validation experiments showed that the transition matrix for chains of order 3 gives the best performance for most datasets (though some datasets produce better cross validation results when using different orders). Based on this, we used Markov chains of order 3 for the rest of our experiments with the Markovian distance.

Figure 4(a) shows that the Markovian method with failure-prediction outperforms the Euclidean distance in 20 datasets. Moreover, switching to the Markovian distance only when Euclidean truly fails (i.e., failure prediction with a perfect oracle) results in a significant improvement in almost all the datasets (refer to Figure 4(b)). This improvement signifies that the Markovian distance is able to capture semantics embedded in the series, which the Euclidean distance fails to uncover.

Finally, we turn our attention to the LDD distance. This method uses a series of distances calculated using a sliding window over the query series Q and each series X_i in the dataset. The distribution of the resulting sliding window based distances is represented as a histogram. We then calculate the joint probability of each X_i being the nearest neighbor (i.e., the corresponding LDD value is the smallest). Maximizing this probability gives us the most probable class C_i for a query Q. The sliding window sizes were set independently for each dataset, and were selected during the training phase by maximizing F1.

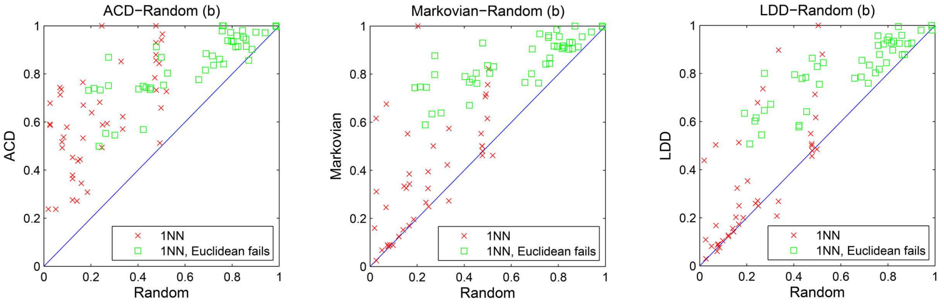
Figure 5(a) depicts the results of the comparison between the combination of LDD with Euclidean (i.e., LDD is used when Euclidean is predicted to fail), and Euclidean. As with the other two proposed measures, the methodology that uses the LDD distance is able to outperform Euclidean in some, but not all datasets we tested. Once again, when the failure of the Euclidean distance based classifier can be perfectly predicted, then the advantage of switching to the LDD measure is significant for all datasets.

5. CONCLUSIONS

In this work, we argued about the utility of taking into account the correlations inherent among neighboring values of a sequence, when designing distance measures for data series. We proposed three different measures that are correlation aware, based on autocorrelation, Markov chains, and the subsequence distance distributions.

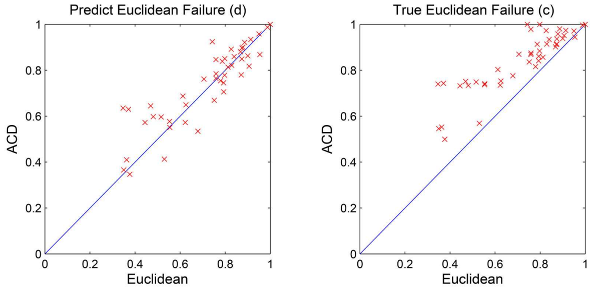
Our preliminary experimental results with 43 real datasets show that these more complex distance measures have the potential to compute distances more accurately, as demonstrated using the 1NN classification results. This result is explained by the fact that they can effectively encode information about the sequentiality of the points in a data series, which is completely ignored by the Euclidean distance.

In our future work, we plan to conduct more detailed experiments for the characterization of the performance behavior of the proposed distances, as well as new ones. Moreover, we will study in depth the problem of when to use the correlation-aware measures, and how to combine them with other distance measures. This proves to be a critical



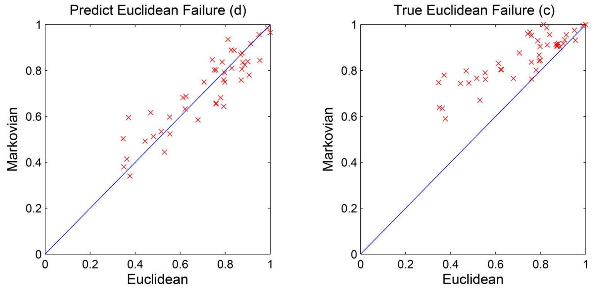
(a) (b) (c)

Figure 2: Comparison of ACD, Markovian, and LDD to a random classifier



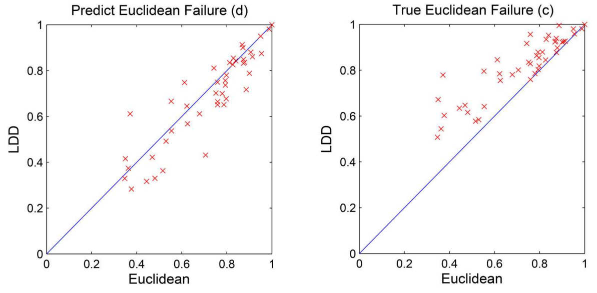
(a) (b)

Figure 3: Comparison for ACD distance



(a) (b)

Figure 4: Comparison for Markovian distance



(a) (b)

Figure 5: Comparison for LDD distance

step in order to exploit the benefits of the proposed distance measures.

Acknowledgements

We would like to thank Muhammad Usman Akram, who contributed in the implementation and experimental evaluation of the techniques described in this paper.

References

[1] D. J. Berndt and J. Clifford. Using dynamic time warping to find patterns in time series. In *AAAIWS*, pages 359–370, 1994.

[2] A. Camerra, T. Palpanas, J. Shieh, and E. Keogh. isax 2.0: Indexing and mining one billion time series. In *ICDM*, 2010.

[3] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. Keogh. Beyond one billion time series: indexing and mining very large time series collections with isax2+. *KAIS*, 39(1):123–151, 2014.

[4] M. Dallachiesa, B. Nushi, K. Mirylenka, and T. Palpanas. Similarity matching for uncertain time series: Analytical and experimental comparison. QUeST '11, pages 8–15. ACM, 2011.

[5] M. Dallachiesa, B. Nushi, K. Mirylenka, and T. Palpanas. Uncertain time-series similarity: return to the basics. *Proceedings of the VLDB Endowment*, 5(11):1662–1673, 2012.

[6] M. Dallachiesa, T. Palpanas, and I. F. Ilyas. Top-k nearest neighbor search in uncertain data series. *PVLDB*, 8(1):13–24, 2014.

[7] G. Das, D. Gunopulos, and H. Mannila. Pkdd. *Principles of Data Mining and Knowledge Discovery*, pages 88–100, 1997.

[8] B. Dasarathy. Nearest Unlike Neighbor (NUN): An Aid to Decision Confidence Estimation. In *Optical Engineering 34*, 1995.

[9] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *Proceedings of the VLDB Endowment*, 1(2):1542–1552, 2008.

[10] K. Kalpakis, D. Gada, and V. Puttagunta. Distance measures for effective clustering of arima time-series. In *ICDM*, pages 273–280, 2001.

[11] E. Keogh, X. Xi, L. Wei, and C. Ratanamahatana. The UCR Time Series Classification/Clustering Homepage, 2011.

[12] A. Kotsifakos, V. Athitsos, and P. Papapetrou. Query-sensitive distance measure selection for time series nearest neighbor classification. *IDA*, 20(1):5–27, 2016.

[13] K. Mirylenka, V. Christophides, T. Palpanas, I. Pefkianakis, and M. May. Characterizing home device usage from wireless traffic time series. In *EDBT*, pages 539–550, 2016.

[14] K. Mirylenka, G. Cormode, T. Palpanas, and D. Srivastava. Conditional heavy hitters: detecting interesting correlations in data streams. *The VLDB Journal*, 24(3):395–414, 2015.

[15] K. Mirylenka, T. Palpanas, G. Cormode, and D. Srivastava. Finding interesting correlations with conditional heavy hitters. In *ICDE*, pages 1069–1080, 2013.

[16] T. Palpanas. Data series management: The road to big sequence analytics. *SIGMOD Record*, 44(2):47–52, 2015.

[17] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *KDD*, pages 262–270, 2012.

[18] K. Zoumpatianos, S. Idreos, and T. Palpanas. ADS: the adaptive data series index. *VLDB J.*, 25(6):843–866, 2016.

Load balancing for Key Value Data Stores

Ainhoa Azqueta-Alzúaz
Marta Patiño-Martínez
Universidad Politécnica de Madrid
Madrid, Spain
{aazqueta,mpatino}@fi.upm.es

Ivan Brondino
Ricardo Jimenez-Peris
LeanXcale
Madrid, Spain
{ivan.brondino,rjimenez}@leanxcale.com

ABSTRACT

In the last decade new scalable data stores have emerged in order to process and store the increasing amount of data that is produced every day. These data stores are inherently distributed to adapt to the increasing load and generated data. HBase is one of such data stores built after Google BigTable that stores large tables (hundreds of millions of rows) where data is stored sorted by key. A region is the unit of distribution in HBase and is a continuous range of keys in the key space. HBase lacks a mechanism to distribute the load across region servers in an automated manner. In this paper, we present a load balancer that is able to split tables into an appropriate number of regions of appropriate sizes and distribute them across servers in order to attain a balanced load across all servers. The experimental evaluation shows that the performance is improved with the proposed load balancer.

Keywords

Big Data, Key Value, HBase, Load Balancing, Performance

1. INTRODUCTION

During the last years new scalable data stores have emerged in order to process and store the increasing amount of data that is produced every day. These data stores also known as NoSQL data stores remove most of the relational databases properties in order to achieve high scalability. These data stores are inherently distributed to adapt to the increasing load and generated data. HBase [2] is one of such data stores built after Google BigTable that stores large tables (hundreds of millions of rows) where data is stored sorted by key. Each table defines a set of column families and a column can be defined at any time (two rows may have different columns). Data in HBase is organized into regions, which are the unit of distribution. A region is a continuous range of keys in the key space. HBase provides mechanisms for load balancing across servers by moving regions among servers. However, regions remain unchanged. That is, if a region becomes a hotspot, the HBase load balancing mechanisms will not distribute the region among the servers. That region will be managed by a single server. Another example where this may happen is the case when a region hosts much more keys than other regions (it manages more keys) and most of the load targets that region. Paper [4] proposes a load balancing algorithm for HBase. The algorithm moves regions across the servers to balance the load however, if a region becomes a hot spot and most of the load (for instance, 90% of the load) targets that region, moving the region would not balance the load among the servers. The only way to balance the load is to partition that region into smaller regions which then, will be moved to consecutive servers.

In this paper we target the partition of regions into regions hosting a similar number of rows. This policy is effective for those situations where a region becomes overloaded and by applying the predefined load balancing mechanisms the situation cannot improve. Our preliminary results loading the database defined by TPC-C benchmark for 3000 warehouses in a cluster of ten servers for storing data show that the throughput is increased one order of magnitude and the latency decreases two orders of magnitude.

The rest of the paper is organized as follows. Section 2 presents an introduction to HBase. Section 3 describes the proposed data partitioning algorithm. Section 4 describes the performance evaluation and the cost of the proposed approach. Finally, conclusions are presented in Section 5.

2. HBASE

HBase [2] is a sparse distributed scalable key-value data store modelled after Google BigTable [1].

HBase organizes data in very large tables with billions of rows and millions columns. Rows are uniquely identified by a key. Columns are organized into column families, which are defined at the time a table is created. Columns can be defined at any time and can vary across rows. A cell is a combination of {*key, column family, column*} and contains a *value* and a *timestamp* which represents the value version. Timestamps are automatically defined or can be user defined. For instance, the cell {customerid, address:home} references the last provided home address of the *customerid*, which is stored in the column family *address* and column *home*. Keys are bytes and rows are sorted alphabetically based on their key.

Tables are distributed in a cluster through regions. *Re-*

Series ISSN: 2367-2005

10.5441/002/edbt.2017.56

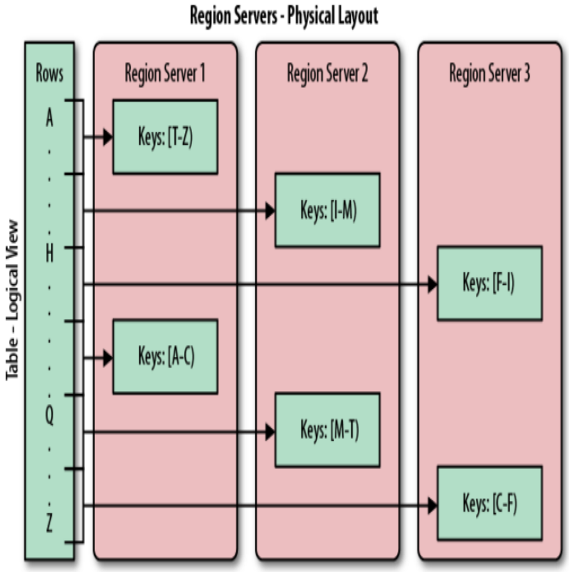


Figure 1: Rows group in regions and served by different servers [2]

gions are defined by key ranges. Regions can be automatically split by HBase or manually by defining the start key of a region. A *Region Server* manages the regions of a server. Regions are automatically split into two regions when they reach a given size or using a custom policy. Manual splitting of regions can be done at table creation time (*pre-splitting*) or later. This is advisable for instance when a hotspot is created on a region. By partitioning the region, the data can be handled by two or more servers.

Figure 1 shows an HBase deployment with three servers (pink boxes), each one hosting one region server. The keys in the table rows range from A to Z (on the left). Each region server handles two regions. Region server 1 handles keys in ranges T to Z and A to C. By default, tables have a unique region when they are created. A region is split into two regions automatically when it reaches a given limit. There are several predefined split policies, which basically split a region when the associated file reaches a given size or based on the number of regions a region server hosts. A table can be pre-split into regions either when it is created or later providing the key ranges.

Internally, the HBase Master stores metadata for instance, the location of the different regions of a table. The actual data of a region (keys and associated information) is stored in HFiles. There are as many HFiles as column families a table has. HFiles are stored in HDFS [3] to achieve high availability.

2.1 HBase Load Balancing Algorithms

HBase provides an automatic load balancer that runs on the master and distributes regions on the cluster every five minutes by default. HBase load balancer implements three algorithms [2]:

- *Simple Load Balancer.* This algorithm takes into account the number of regions each region server is managing and the load at each server. The goal is that all region servers will handle a similar number of regions by moving regions from the more loaded servers to the least loaded regions servers.
- *Favored Node Load Balancer.* This load balancing algorithm assigns favoured server for each region. The primary region server hosts the region. There are also secondary and tertiary region servers. HDFS uses the favoured servers information for creating HDFS files and placing the blocks of the file. When the primary region server crashes, the secondary takes over providing low latencies.
- *Stochastic Load Balancer.* This algorithm searches a region distribution that minimizes a cost function. This function is computed taking into account the region load, table load, data locality, MemStore sizes and HFile sizes. This algorithm has several parameters, for instance, to control the maximum number of regions to be moved, minimize the number of times the balancer will try to mutate all servers.

None of these HBase load balancers changes the region configuration. They move regions among servers to distribute the load. However, if there are several regions that become hotspots, these regions will not be split and distributed among region servers to distribute their load.

This is the goal of the Static Load Balancer we present in the next section. The load balancer distributes the keys of a table among regions in order to ensure that each region contains the same amount of keys and distributes the regions among all region servers.

3. STATIC LOAD BALANCER

The static load balancer goal is to create regions with a similar number of rows and distribute them across all regions servers in a cluster. For this purpose, the load balancer needs the table size and the key distribution. The algorithm generates a set of keys that define the new regions with the same number of keys. Then, it splits current regions according to the new regions provided by the load balancer and assigns them uniformly among region servers.

3.1 Table Histogram

In order to divide a table into regions managing a similar number of keys, the total number of rows of the table and the stored keys are needed. The table histogram scans regions reading every x number of rows (for instance, every 1000 rows). For every x rows, it stores the key of that row. For instance, if a region hosts 2500 rows and $x = 1000$, it will store three keys, 1000, 2000 and 2500. The values associated to those keys will be the keys that are stored in positions 1000, 2000 and 2500, respectively in that region. The histogram runs as an HBase coprocessor [2] so, no data is moved outside the server hosting the region.

The histogram information is used to calculate the number of rows in the table, $\#RowsTable$, the number of rows each region is currently handling, $\#RowsRegion$, the expected number of rows per region, $\#ExpectedRowsRegion$, the total number of rows that are wrongly placed, $\#WrongPlacedRows$ and the standard deviation of rows wrongly placed, $\%STDofWrongPlacedRows$. This value is used to decide

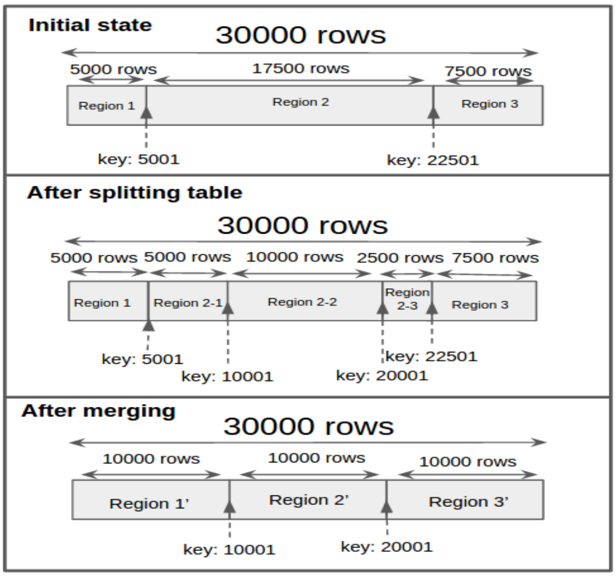


Figure 2: Load Balancer Example

whether the static load balancer should be executed. Currently, the system administrator defines a threshold, $\%STDThreshold$. If the standard deviation is greater than the threshold, the load balancer is executed.

3.2 Load Balancer

The load balancer (Algorithm 1) uses the information generated by the histogram for defining the new regions. Given the expected number of rows per region, $\#ExpectedRowsRegion$, the load balancer obtains the split points of the region by traversing the histogram. The key stored every $\#ExpectedRowsRegion$ positions will define the new regions.

For instance, Figure 2 shows a table with 30000 keys. Initially there are three regions, Region 1, Region 2 and Region 3, which handle 5000, 17500 and 75000 rows, respectively. Region 1 handles keys from 0 up to 5000, Region 2 manages the keys in the range 5001 and 22500 and so on. The final distribution the load balancer will define consists of three regions each one managing 10000 keys (each region will store a similar number of keys). If the histogram stores the keys every 10000 rows ($histogramPrecision$), the $splitPoints$ will be the keys stored at position 10001 and 20001.

Then, the algorithm splits the regions using HBase *HBaseAdmin.split()* method proving the split points. At this point the previous regions and the new ones coexists. For instance, there are 5 regions in the example in Figure 2-*After splitting table*. That is, the old regions and the new ones coexists Region 2 is split into three regions, Region 2-1, Region 2-2 and Region 2-3, with 5000, 10000 and 2500 rows each one. Only Region 2-2 will be a final region after the load balancing finishes. The other two regions will be merged with Region 1 and Region 3, respectively in order to achieve the three final regions with the same number of rows (Region 1', Region 2' and Region 3') (Figure 2-*After merging*).

As a final step in Algorithm 1 the location of the regions is stored in the Zookeeper instance running on HBase (*RegionsLocation*). This step avoids that if HBase stops and starts, by default, the regions are assigned randomly to region servers and then, the data files need to be moved to the new server where the region is handled.

Algorithm 1 Load Balancing

```
Require: $table$, $stdThreshold$
$histogramPrecision = 10000$
$generateHistogram(table)$
$\#RS \leftarrow get\#RegionServers()$
$\#Regions \leftarrow get\#Regions(table)$
$\#RowsTable \leftarrow get\#RowsTable(table)$
$\#ExpectedRowsRegion \leftarrow get\#ExpectedRowsRegion(table)$
$\%STDofWrongPlacedRows \leftarrow getSTDofWrongPlacedRows(table)$
if $\%STDofWrongPlacedRows > \%STDThreshold$ then
    $splitPoints \leftarrow getNewSplitPoints(histogramPrecision, \#RowsTable, \#RS, \#ExpectedRowsRegion)$
    $split(splitPoints)$
    $merge(splitPoints)$
    $majorCompact(table)$
    $RegionsLocation(table)$
end if
```

4. PERFORMANCE EVALUATION

In this section we present the performance evaluation of the proposed load balancer. The evaluation has been conducted in a cluster of 11 nodes; each node is 64 core AMD Opteron 6376 @ 2.3GHz, equipped with 128GB of RAM, 1Gbit Ethernet and a direct attached SSD hard disk of 480GB running Ubuntu 12.04.5 LTS. One of the nodes is used for hosting metadata servers, HDFS NameNode, HBase Master and ZooKeeper. The rest of nodes are used as worker nodes, each one running one HDFS Data Node and four HBase-Region Servers. That is, there are 10 DataNodes and 40 RegionServers. We use the Cloudera distribution of Apache HBase with version CDH5.3.5.

The load balancer is evaluated loading the data defined by TPC-C benchmark since there are different tables with different number of columns and different number of rows. The benchmark defines 9 tables. The number of warehouses defines the sizes of the tables. In this initial evaluation, the number of warehouses is 3000. The smallest table holds 3000 rows and the largest 765 million of rows. In order to evaluate the benefits of the proposed load balancing algorithm we evaluate the performance of HBase using TPC-C with the tables split into regions with a random size. Then, we evaluate the performance of the benchmark when the regions handle a similar number of rows.

The unbalanced configuration is presented in Table 1, which shows for each table the total number of rows (*#Rows*), the number of rows of the smallest and largest regions (*#Rows Small Region and #Rows Large Region*) and the standard deviation of the rows that are wrongly placed for each table (*#STD*). For instance, warehouse and order_line tables are the smallest and largest tables, respectively.

Order_line table stores 765 million of rows. The smallest region for that table stores 11181 keys while the largest hosts 1214955735 rows (1212 millions of rows).

Table 1: Data Distribution Before Load Balancing

Table	#Rows	#Rows small region	#Rows large region	#Rows STD
warehouse	3000	6	234	63
district	30000	48	3267	795
item	100000	2	9388	22159
new_order	27M	17749	2837997	619998
orders	90M	33865	9721682	2306951
ix_orders	90M	22865	9721682	2306951
history	90M	2105753	2463901	172493
customer	90M	24592	7754305	2206718
ix_customer	90M	24592	7754305	2206821
stock	300M	18376	31182657	7463525
ix_stock	300M	113700	258520000	2509062
order_line	765M	11181	121495735	87312734
ix_order_line	765M	557433	84431120	97324041

Table 2: Data Distribution After Load Balancing

Table	#Rows	#Rows small Region	#Rows large Region	#Rows STD
warehouse	3000	75	75	0
district	30000	750	750	0
item	100000	2500	2500	0
new_order	27M	500000	684408	28020
orders	90M	1980000	2260000	743379
ix_orders	90M	1980000	2260000	43379
history	90M	2106778	2462829	172144
customer	90M	1975165	2260000	44120
ix_customer	90M	1975165	2260000	18502
stock	300M	7202284	7510000	47762
ix_stock	300M	7152000	7512000	55815
order_line	765M	18940000	19134682	29743
ix_order_line	765M	18820457	19140014	48984

4.1 Load Balancer Evaluation

In this section we present how the load balancer distributes the keys into regions given the previous distribution of data. Then, we evaluate the performance of TPC-C with both configurations and finally, we present the time for executing the load balancing algorithm.

Table 2 shows the size of the smallest region (the one hosting less keys) and the largest one for each table after running the static load balancer. The results show that the difference in number of keys hosted by these regions is less than 1%. The smallest region of table Order_line now stores 18940000 rows and the largest one stores 19134682 rows that is, 18.9 million rows and 19.1 rows respectively. We can compare those results with the ones in Table 1, which produced for the same Order_line a region with 11181 keys, while the largest region hosts 1214 millions of rows.

Table 3 shows results in terms of throughput, in transactions per minute, and latency of transactions, in milliseconds, of running TPC-C benchmark with the unbalanced and balanced regions distribution. The throughput of TPC-C with the unbalanced regions reaches 3296 transactions with an average response time of 1550.805 ms. When the regions have a similar size (i.e., after running the static load balancer), the throughput is multiplied by 10, processing 36761 transactions per minute with an average response time of 16.858 ms. That is, the response time is two orders of magnitude lower.

Finally, the execution time of the load balancer for each table of TPC-C is shown in Table 4. Table *History* is not balanced by the Static Load Balancer because it is already balanced (i.e., the $\%STDofWrongPlacedRows$ is below than 1%).

Table 3: TPC-C Execution

	Before Load Balancer	After Load Balancer
Throughput (tpmCs)	3296	36761
Avg. Latency (ms)	1550.805	16.858

Table 4: Load Balancer Execution Times

Table	#Rows	Histogram	Split Table	Merge Regions	Regions Location
warehouse	3000	00:00:02.009	00:00:06.453	00:00:11.356	00:00:10.366
district	30000	00:00:02.754	00:00:08.484	00:00:09.687	00:00:10.366
item	100000	00:00:01.946	00:00:08.070	00:00:10.022	00:00:10.400
new_order	27M	00:00:21.037	00:00:46.137	00:00:17.196	00:00:10.344
orders	90M	00:02:56.017	00:07:50.062	00:01:12.132	00:00:10.372
ix_orders	90M		00:07:29.129	00:00:44.998	00:00:10.245
history	90M	00:00:31.911			00:00:10.380
customer	90M	00:03:11.329	00:12:35.812	00:04:23.994	00:00:10.402
ix_customer	90M		00:12:41.747	00:03:11.593	00:00:10.251
stock	300M	00:10:19.886	00:39:10.252	00:07:12.418	00:00:10.312
ix_stock	300M		00:09:10.791	00:04:41.104	00:00:10.328
order_line	765M	00:37:53.463	01:59:57.023	00:11:23.552	00:00:10.421
ix_order_line	765M		00:29:52.061	00:26:18.463	00:00:10.316

Most of the time is spent in the split process, which divides regions into several regions. Each time a region is split, a major compact process is executed in order to split the stored files (HFiles) into two. This process is very expensive for large tables (more than 100 million rows).

5. CONCLUSIONS

In this paper we have presented a Load Balancer algorithm that partitions regions into regions that manage a similar number of keys. The performance evaluation shows that this greatly improves performance. However, the execution of the load balancer is time consuming. This process should be run seldom during off-peak periods. Fault tolerance for the algorithm remains as future work.

6. ACKNOWLEDGMENTS

This research has been partially funded by the European Commission under projects CoherentPaaS, LeanBigData and CloudDBApplience (grants FP7-611068, FP7-619606 and H2020-732051), the Madrid Regional Council, FSE and FEDER, project Cloud4BigData (grant S2013TIC2894), the Ministry of Economy and Competitiveness (MINECO)under projects BigDataPaaS and CloudDB (grant TIN2013-46883 and TIN2016-80350).

7. REFERENCES

[1] 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 Transactions on Computer Systems*, 26(2):4:1–4:26, June 2008.

[2] L. George. *HBase: The Definitive Guide*. O'Reilly Media, 2011.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[4] L. Xia, H. Chen, and H. Sun. An optimized load balance based on data popularity on hbase. In *2nd International Conference on Information Technology and Electronic Commerce (ICITEC)*, pages 234–238, Dec 2014.

Poster Paper

Entity Matching on Web Tables: a Table Embeddings Approach for Blocking

Anna Lisa Gentile
University of Mannheim
annalisa@informatik.uni-mannheim.de

Petar Ristoski
University of Mannheim
petar.ristoski@informatik.uni-mannheim.de

Steffen Eckel
University of Mannheim
steffen.eckel@students.uni-mannheim.de

Dominique Ritze
University of Mannheim
dominique@informatik.uni-mannheim.de

Heiko Paulheim
University of Mannheim
heiko@informatik.uni-mannheim.de

ABSTRACT

Entity matching, or record linkage, is the task of identifying records that refer to the same entity. Naive entity matching techniques (i.e., brute-force pairwise comparisons) have quadratic complexity. A typical shortcut to the problem is to employ blocking techniques to reduce the number of comparisons, i.e. to partition the data in several blocks and only compare records within the same block.

While classic blocking methods are designed for data from relational databases with clearly defined schemas, they are not applicable to data from Web tables, which are more prone to noise and do not come with an explicit schema. At the same time, Web tables are an interesting data source for many knowledge intensive tasks, which makes record linkage on Web Tables an important challenge. In this work, we propose an unsupervised approach to partition the data, that does not exploit any external knowledge, but only relies on heuristics to select the blocking attributes. We compare different partitioning methods: we use (i) clustering on bag-of-words, (ii) binning via Locality-Sensitive Hashing and (iii) clustering using word embeddings. In particular, the clustering methods show good results on a standard dataset of Web Tables, and, when combined with word embeddings, are a robust solution which allows for computing the clusters in a dense, low-dimensional space.

Keywords

Instance matching; Web tables; blocking methods; word embeddings

1. INTRODUCTION

Entity matching aims at identifying different descriptions in unstructured or (semi-)structured textual content that refer to the same real-world entity. Similarly, the deduplication task (or record linkage) looks for nearly duplicate records in relational data, usually amongst data points that refer the same entity type. In both cases, if addressed in a brute force fashion, the matching task has quadratic complexity, as it requires pairwise comparisons of all the records. The most widely adopted solution for this problem is to group records in blocks before comparing them, a pre-processing step usually referred to as *blocking.* Blocking offers a compromise between the number of comparisons to perform and the number of missed entity matches [3].

Traditionally, a plethora of blocking techniques have been proposed to reduce number of comparisons [5, 6, 12, 13], especially for the deduplication task. These exploit specific clues from the data schema, ad hoc similarity functions and mapping rules, as well as background knowledge of the domain and of the type of the entities.

In this paper, we focus on Web tables (i.e., tables on HTML pages), which stand in the middle ground between unstructured Web content and relational data. From a structural point of view records in Web tables resemble records in database tables. Nevertheless, they come with no schema attached, making the direct usage of traditional blocking techniques not applicable, since we cannot rely on any type of assumption for the data nor have domain knowledge [3]. Thus, in this scenario, the high number of entity types and the representational heterogeneity, even for entities of the same semantic type, make traditional blocking techiques hard to apply. Therefore, we argue for the need of a domain agnostic representation of Web tables, which has the property of being succinct and can be used in a similar fashion as the signatures in the traditional blocking techniques. We explore different heuristics based on bag of words and word embeddings, combined with different clustering methods.

By using a publicly available gold standard (Section 4.1), we measure the pair comparison reduction ratio and the pair completeness of the instance matching task, when performed after the blocking step. We show that the blocking based on table embeddings leads to the best tradeoff between (i) reduction ratio on the number of pair comparisons to perform and (ii) pair completeness (recall on entity matches). Moreover, it is robust with respect to the type of table preprocessing (Section 3.1), while also offering a succinct representation for the tables.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.57

The main contribution of this work is twofold. First, we propose a novel solution to represent tables in a latent space using Neural Language Models (NLM). NLM have been proved successful in replacing the classical bag of word representation of text (binary feature vector, where each vector index represents one word) with a latent representation with a lower vector dimensionality. Second, we exploit the latent table representations to perform blocking in the context of entity matching in Web tables.

2. STATE OF THE ART

Entity matching is a task with quadratic complexity, therefore, when applied to larger data collections, it is necessary to reduce the number of comparisons to be performed. A common way to reduce complexity is the use of so-called *blocking strategies* to reduce the search space [13] and achieve a reasonable compromise between the number of comparisons and the number of missed entity matches [3]. Traditionally, blocking techniques exploit specific criteria in the data schema to split the data before performing entity comparisons, only utilizing the values of some key attributes. A typical example is the Sorted Neighborhood Method (SNM) [5], which performs sorting of the records according to a specifically chosen Blocking Key.

Content based blocking strategies usually look for common tokens between two entities. The search for common tokens can be performed in entities descriptions or it can be restricted to the values of attributes that overall have similar values [12]. In a naive scenario, each token defines a new block and all entities that share the token end up in that block (here, entities may belong to multiple blocks); more sophisticated methods index and rank tokens, so that the search is restricted to the n most frequent ones [8]. Another commonly used technique is *Locality-Sensitive Hashing (LSH)*, which produces effective signatures of records to perform fast comparisons amongst entities. Specifically, Duan et al. [4] used LSH to perform instance based ontology matching. The underlying assumption of their work is that discovering relationships in data from different sources can only be achieved after correctly typing the data. Their objective is to perform comparisons amongst *entity types*. The representation of each type is the sum of all its entities, which can be seen as a big document, where they can exploit the tf-idf weighting schema to select relevant tokens.

Multi-Block is a blocking strategy that uses a multidimensional index in which similar objects are located near each other. In each dimension, the entities are indexed by a different property to achieve effect retrieval [6]. To boost recall, data is enriched by interlinking to DBpedia [9].

Differently from available state of the art we propose a strategy which is (i) completely agnostic w.r.t. the data schema, (ii) does not rely on external knowledge and (iii) performs only minimal preprocessing of the data.

3. A BLOCKING APPROACH USING TABLE EMBEDDINGS

The blocking step is performed at table label. We propose an adaptation of neural language models (NLM) to represent table embeddings, which provide a succinct latent representation of tables without relying on any domain knowledge. In the following, we describe the preliminary table preprocessing step, followed by the proposed embedding strategy.

3.1 Table Preprocessing

As Web tables are different from relational tables, i.e., they are schema-free and prone to noise, preprocessing and normalization are required. We reuse components from the "Mannheim Search Join Engine" [10] to: (i) identify pseudo key attributes (the subject column); (ii) recognize table header structures; and (iii) identify data types.

To identify the subject column we apply the heuristic proposed by [14] of choosing the column of type string with the highest number of unique values. In case of a tie, the leftmost column is used. The subject column basically contains entity names which act as pseudo-keys for the table [2, 15, 16]. In the example depicted in Table 1, the first column is used as a subject column.

For detecting the headers, we assume that the header row is the first non-empty row. In the example depicted in Table 1, the first row is used as headers.

For data type detection, we use about 100 manually defined regular expressions to detect numeric values, dates, and links. Based on the data type, the values of each column are normalized, e.g., the string values are lower-cased and special characters are removed. Furthermore, for this work we replace numbers, timestamps, and geo-coordinates with a static value, as our approach currently does not handle numeric values. For example, given Table 1, the first two columns will be recognized as strings, and the last two columns as numerical. The first row will be recognized as the header row, and the first column as the subject attribute. After preprocessing, the table will look like Table 2.

Table 1: Example table of countries

Country	Capital	Population	GDP (USD)
Germany	Berlin	80M	46,268.64
France	Paris	60M	42,503.30
United Kingdom	London	64.1M	41,787.47

Table 2: Preprocessed table

h:country	h:capital	h:population	h:gdp_(usd)
v:germany	v:berlin	\$NUM\$	\$NUM\$
v:france	v:paris	\$NUM\$	\$NUM\$
v:united_kingdom	v:london	\$NUM\$	\$NUM\$

3.2 Table Embeddings

NLMs are explicitly built to take into account the order of words in text documents and to encode a stronger statistical dependence amongst words which are closer in the sequence. As the model is originally designed for raw text, where the sequence of words is naturally derivable from the sentences, we first need to transform tables to word sequences, which can be used to train a NLM. We consider table values and table attributes instead of word sequences. Thus, in order to apply such approaches on table data, we first have to transform the tables into sequences of values and attributes, which can be considered as sentences. Using those sentences, we can train the same neural language models to represent each value and attribute in the table as a vector of numerical values in a latent feature space.

We propose three general approaches to transform tables to sentences: (i) *attributes model*: the header row is

converted into a sequence of attributes, e.g., for Table 2 the produced sequence is: *"h:country h:capital h':population h:gdp_(usd)"*; (ii) *entities model*: the subject column is converted into a sequence of entities, e.g., for Table 2 the produced sequence is: *"v:germany v:france v:united_kingdom"*. (iii) *attributes and entities model*: we convert the table into a sequence of triples of the form <entity, header, value>, where the entity is the subject, the header is the predicate, and the corresponding value is the object; e.g. for Table 2 the produced sequence is: *"v:germany h:capital v:berlin; v:germany h:population $NUM$; ... ;*
v:united_kingdom h:gdp_(usd) $NUM$".

As NLM, we use *word2vec* [11] , a two-layer neural net model that learns word embeddings from raw text. Specifically, we employ the skip-gram model, which tries to predict the context words given a target word. Given a sequence of words $w_1, w_2, w_3, ..., w_T$ and a context window c, the objective of the skip-gram model is to maximize the following average log probability:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0} logp(w_{t+j}|w_t), \tag{1}$$

The probability $p(w_{t+j}|w_t)$ is calculated using the softmax function:

$$p(w_o|w_i) = \frac{exp(v'^{T}_{wo}v_{wi})}{\sum_{w=1}^{V} exp(v'^{T}_{w}v_{wi})}, \tag{2}$$

where v_w and v'_w are the input and the output vector of the word w, and V is the entire vocabulary.

Directly calculating the softmax function is inefficient, as it is proportional to the size of V, therefore we used *negative sampling* as an optimization technique, following the approach discussed in [11].

Once the training is finished, all words (i.e., table values and attributes in our case) are projected into a lower-dimensional feature space, and semantically similar words are positioned close to each other. We can then use the produced latent representation of tables to calculate the similarity between two tables using the standard cosine similarity measure.

4. EXPERIMENTS

The aim of the experiments is to verify how the proposed table embeddings perform in reducing the complexity of entity matching, specifically in reducing the number of performed pair comparisons, without loosing recall. As entity matching per se is not the focus of this paper, we only consider the *pair completeness*, i.e., the number of entity pairs that are contained in the same bucket identified by the blocking mechanism. That way, we can directly evaluate the tradeoff between (i) the reduction ratio achieved with the blocking and (ii) the pair completeness, i.e. the ratio of matches that can be potentially identified.

4.1 The T2D Dataset

As a gold standard, we use the publicly available T2D dataset[1]. T2D contains a subset of the Web Data Commons Web Tables Corpus[2] for which schema-level and instance-level correspondences to DBpedia 2014 are provided. We use the 233 tables of T2D for which rows are mapped to entities in the DBpedia knowledge base (for a total 26,124 entity-level correspondences). As we are not interested in matching to DBpedia as an external knowledge base, but rather finding correspondences *within* a dataset, we transform the matches to internal matches within the tables, i.e., for every pair of table rows mapped to the same DBpedia entity, we create a correspondence amongst the two table rows. This process produces 50,072 instance correspondences.

To build the table embedding models, we use the T2D dataset and the WikiTables dataset[3]. The WikiTables dataset has been extracted from Wikipedia pages in the course of the WikiTables project [1]. The corpus consists of 1.35 million Wikipedia tables. Additionally, we extracted $365,194$ tables from the Wikipedia 2015 dumps. We build Skip-Gram models with the following parameters: window size = 10; number of iterations = 15; negative sampling for optimization; negative samples = 25; 500 latent dimensions. All the models, as well as the code, are publicly available[4].

4.2 Results

Overall, we compared two different representations for tabular data. We use the classical bag of word model (bow) and the table embeddings. We build all the representations using four different input features, as identified in the table pre-processing step (Section 3.1): (i) the full content of the table, (ii) the table header, (iii) the table subject column and (iv) the combination of the table subject column and the table header. For (i) and (iii), we use the attributes and entities model, for (ii), we use the attributes model, and for (iii), we use the entities model, as described in section 3.2. Once the representations are built, we use either unsupervised clustering (k-means) or binning via Locality-Sensitive Hashing (LSH) to partition tables into bins.

Table (3) shows the results of applying k-means to the representations (according to the four different input features) generated via *table embeddings* (shortened in the table as *emb*) and *bag of words* (shortened in the table as *bow*). As we are looking for a good tradeoff between the pair comparison reduction ratio and pair completeness of matched instances, we also report the harmonic mean (*hm*) of those two values for each run. The first observation concerns the input features. In all scenarios the table headers alone are not enough to perform effective binning. The subject column (also combined with the table header) instead leads to good performance, regardless of the representation, with comparable performance (>90%) for 10 and 20 bins. Furthermore, we can observe that the word embeddings are also competitive when using the full table as input, again with performance >90% for 10 and 20 bins. This means that the potentially expensive and error-prone detection of header and key columns can be omitted when using that approach.

We also perform LSH using a java-LSH implementation[5] both with the standard bow model as well as using the table embeddings as input (using the Super-Bit algorithm [7]). While the table embeddings representation outperformed the standard bow, the general performance were quite low (hm around 60%). This is a direct consequence of the size of the gold standard. As we perform the binning at table level and not at instance level, the dataset only contains 233

[1] http://webdatacommons.org/webtables/goldstandard.html
[2] http://webdatacommons.org/webtables/
[3] http://downey-n1.cs.northwestern.edu/public/
[4] http://data.dws.informatik.uni-mannheim.de/table2vec/
[5] https://github.com/tdebatty/java-LSH

Table 3: Pair comparison reduction ratio (rr) and pair completeness (pc) of the instance matching task and their harmonic mean (hm), when performed after blocking using either table embeddings (emb) or simple bag of word model (bow), when run on either the full content of the table (all), the table header (h), the table subject column (k) or the table subject column and the table header (k+h).

input	#k	rr-emb	pc-emb	hm-emb	rr-bow	pc-bow	hm-bow
all	5	0.75	0.98	0.85	0.74	0.89	0.81
all	10	0.85	0.98	**0.91**	0.83	0.5	0.62
all	20	0.91	0.97	**0.94***	0.92	0.5	0.65
all	30	0.93	0.62	0.74	0.91	0.29	0.44
all	40	0.92	0.85	0.88	0.92	0.4	0.56
all	50	0.95	0.65	0.77	0.95	0.32	0.48
h	5	0.79	0.8	0.79	0.73	0.47	0.57
h	10	0.87	0.66	0.75	0.87	0.42	0.57
h	20	0.94	0.38	0.54	0.91	0.41	0.57
h	30	0.96	0.33	0.49	0.93	0.26	0.41
h	40	0.96	0.32	0.48	0	1	
h	50	0.19	0.97	0.32	0	1	
k+h	5	0.66	0.98	0.79	0.79	0.99	0.88
k+h	10	0.85	0.98	**0.91**	0.85	0.99	**0.91**
k+h	20	0.9	0.97	**0.93**	0.91	0.96	**0.93***
k+h	30	0.92	0.96	**0.94***	0.93	0.68	0.79
k+h	40	0.93	0.79	0.85	0.94	0.69	0.8
k+h	50	0.95	0.63	0.76	0.96	0.41	0.57
k	5	0.79	0.99	0.88	0.77	0.92	0.84
k	10	0.87	0.98	**0.92**	0.88	0.98	**0.93***
k	20	0.92	0.97	**0.94***	0.92	0.88	**0.9**
k	30	0.94	0.89	**0.91**	0.93	0.78	0.85
k	40	0.94	0.89	**0.91**	0.93	0.7	0.8
k	50	0.96	0.52	0.67	0.96	0.51	0.67

tables. In constrast, LSH is know to be effective for large datasets, where the number of points in each bin is at least 100.

5. CONCLUSIONS AND FUTURE WORK

The paper presents a method to perform entity blocking for the task of entity matching in Web tables, by representing tables in a latent space using Neural Language Models. When compared to state of the art blocking methods table embeddings prove to be a promising solution. The current study has been performed on a set of 233 Web tables, manually annotated with entities matches (for a total of 50,072 instance correspondences). In future work, we plan to repeat the experiment on a larger dataset of tables from the Web Data Commons project by semi-automatically generating the gold standard, and to explore ways of meaningfully exploiting other value types, such as numbers and dates.

Acknowledgments

This work is part of the project DS4DM (Data Search for Data Mining) http://ds4dm.de, funded by the German Federal Ministry of Education and Research (BMBF).

6. REFERENCES

[1] C. S. Bhagavatula, T. Noraset, and D. Downey. Methods for exploring and mining tables on wikipedia. In *Proceedings of the ACM SIGKDD Workshop on Interactive Data Exploration and Analytics*, IDEA '13, pages 18–26. ACM, 2013.

[2] M. J. Cafarella, A. Halevy, and N. Khoussainova. Data Integration for the Relational Web. *Proc. VLDB Endow.*, 2:1090–1101, 2009.

[3] V. Christophides, V. Efthymiou, and K. Stefanidis. *Entity Resolution in the Web of Data.* MORGAN & CLAYPOOL PUBLISHERS, 2015.

[4] S. Duan, A. Fokoue, and O. Hassanzadeh. Instance-Based Matching of Large Ontologies Using Locality-Sensitive Hashing. pages 49–64, 2012.

[5] M. A. Hernandez and S. J. Stolfo. The merge/purge problem for large databases. In *CM SIGMOD international conference on Management of data, SIGMOD '95.* ACM, 1995.

[6] R. Isele, A. Jentzsch, and C. Bizer. Efficient Multidimensional Blocking for Link Discovery without losing Recall. (WebDB), 2011.

[7] J. Ji, J. Li, S. Yan, B. Zhang, and Q. Tian. Super-Bit Locality-Sensitive Hashing. *Advances in Neural Information Processing Systems2012*, pages 1–9.

[8] B. Kenig and A. Gal. MFIBlocks : An effective blocking algorithm for entity resolution. 38:908–910, 2013.

[9] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. DBpedia – A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia. *Semantic Web Journal*, 2013.

[10] O. Lehmberg, D. Ritze, P. Ristoski, R. Meusel, H. Paulheim, and C. Bizer. The mannheim search join engine. *Web Semant.*, 35(P3):159–166, Dec. 2015.

[11] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[12] G. Papadakis, E. Ioannou, T. Palpanas, C. Niedere, and W. Nejdl. A Blocking Framework for Entity Resolution in Highly Heterogeneous Information Spaces. 25(12):2665–2682, 2013.

[13] V. Rastogi, N. Dalvi, and M. Garofalakis. Large-scale collective entity matching. *Proceedings of the VLDB Endowment*, 4(4):208–218, 2011.

[14] D. Ritze, O. Lehmberg, and C. Bizer. Matching html tables to dbpedia. In *Proc. of the 5th International Conference on Web Intelligence, Mining and Semantics*, WIMS '15, pages 10:1–10:6, 2015.

[15] P. Venetis, A. Halevy, J. Madhavan, M. Paşca, W. Shen, F. Wu, G. Miao, and C. Wu. Recovering Semantics of Tables on the Web. *Proc. of VLDB Endow.*, 4(9):528–538, 2011.

[16] M. Yakout, K. Ganjam, K. Chakrabarti, and S. Chaudhuri. InfoGather: Entity Augmentation and Attribute Discovery by Holistic Matching with Web Tables. In *Proc. of the 2012 ACM SIGMOD Int. Conf. on Management of Data*, pages 97–108, 2012.

Fast Subsequence Search on Time Series Data

Yuhong Li[1]*, Bo Tang[2]*, Leong Hou U[1], Man Lung Yiu[2], Zhiguo Gong[1]

[1]Department of Computer and Information Science, University of Macau
{yb27407,ryanlhu,fstzgg}@umac.mo

[2]Department of Computing, Hong Kong Polytechnic University
{csbtang,csmlyiu}@comp.polyu.edu.hk

ABSTRACT

Many applications on time series data require solving the subsequence search problem, which has been extensively studied in the database and data mining communities. These applications are computation bound rather than disk I/O bound. In this work, we further propose effective and cheap lower-bounds to reduce the computation cost of the subsequence search problem. Experimental studies show that the proposed lower-bounds can boost the performance of the state-of-the-art solution by up to an order of magnitude.

1. INTRODUCTION

The subsequence search problem on time series data has extensive applications in medical diagnosis, speech processing, climate analysis, financial analysis, etc [1, 5]. Specifically, given a query time series q and a target time series t, the subsequence search problem finds a subsequence t_c of t such that it has the smallest distance $dist(q, t_c)$ to q. Figure 1 illustrates the subsequence search problem. The typical distance measures are the Euclidean distance (ED) and Dynamic Time Warping (DTW).

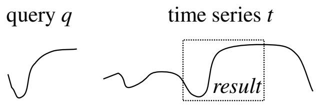


Figure 1: Subsequence search on time series

Euclidean Distance (ED) is the most common similarity measure [1, 5] due to its simplicity and applicability. The distance between two time series of length m is given as follows.

$$ED(q, t_c) = \sqrt{\sum_{i=1}^{m}(q[i] - t_c[i])^2} \quad (1)$$

Dynamic Time Warping (DTW) is proposed to capture the similarity of two sequences which may vary in time or have missing values. It has been shown to be an effective distance measure [2]. This distance is defined as $DTW(q, t_c) = \sqrt{DTW_{SQ}(q, t_c)}$, where $DTW_{SQ}(q, t_c)$ is computed as follows.

$$DTW_{SQ}(q, t_c) = (q[1] - t_c[1])^2 + \min \begin{cases} DTW_{SQ}(q[2...last], t_c) \\ DTW_{SQ}(q[2...last], t_c[2...last]) \\ DTW_{SQ}(q, t_c[2...last]) \end{cases} \quad (2)$$

where $q[2...last]$ denotes the subsequence of q containing values from the 2nd to the last offset. To avoid pathological warping and reduce the quadratic computational cost, many research work [5] suggest to limit the warping length r such that $q[i]$ is matched with $t_c[j]$ if and only if $|i - j| \leq r$. This reduces the complexity of DTW from $O(m^2)$ to $O(mr)$.

To the best of our knowledge, the UCR Suite [5] is the state-of-the-art solution for arbitrary length subsequence search. Since exact distance computations are expensive, the UCR Suite applies a suite of lower-bounds to prune unpromising subsequences, before computing the exact distances for the remaining subsequences. Nevertheless, the subsequence search problem is still a computation intensive problem, especially for increasingly long time series nowadays. Even with the UCR Suite, the subsequence search on a trillion-scale time series would take 3.1 hours (under the Euclidean distance) or 34 hours (under Dynamic Time Warping) on a commodity PC [5].

To reduce the computation time of subsequence search, we propose effective lower-bounds based on the triangle inequality and Piecewise Aggregate Approximation (PAA). The proposed lower-bounds can be computed online and easily integrated into the UCR Suite. According to our experimental evaluations, the proposed methods can improve the performance of the UCR Suite by up to an order of magnitude.

The paper is organized as follows. Section 2 formally defines our problem and presents the state-of-the-art solution (i.e., the UCR Suite). We present our online lower-bounds in Section 3. The experimental study is given in Section 4. Finally, we conclude the paper in Section 5.

2. PRELIMINARIES

2.1 Problem Definition

In this work, we follow the suggestion of UCR Suite [5] that every subsequence must be Z-normalized in order to capture the similarity between the shapes of the sequences. Given a time series t, the Z-normalized value of $t[i]$ can be calculated as: $\hat{t}[i] = \frac{t[i]-\mu_t}{\sigma_t}$,

*indicates equal contribution

Series ISSN: 2367-2005

10.5441/002/edbt.2017.58

where $t[i]$ is the i-th element of t, $\hat{t}[i]$ is the Z-normalized value of $t[i]$, and μ_t and σ_t are the mean and standard deviation of t, respectively.

PROBLEM 1 (SUBSEQUENCE SEARCH PROBLEM). *Given a time series t of length n, a query time series q of length m, and a distance function $dist(\cdot,\cdot)$, the subsequence search problem returns a length-m subsequence $t_c \in t$ such that $dist(\hat{q},\hat{t}_c) \leq dist(\hat{q},\hat{t}_{c'})$, $\forall\ t_{c'} \in t$ of length m.*

A naïve solution for the subsequence search (cf. Problem 1) is to calculate the distance $n - m + 1$ times. The computation of the subsequence search may become prohibitive for long sequences. It should be noted that the search performance is closely related to the distance function $dist(\cdot,\cdot)$.

2.2 The State-of-the-Art: UCR Suite

In the following, we briefly introduce how the UCR Suite can boost ED-based and DTW-based subsequence search.

UCR-ED. For ED-based subsequence search, the UCR-ED [5] first employs the early abandoning technique. In general, it attempts to early terminate the distance computation when the accumulated distance is already larger than the best-so-far distance. To further improve the performance, the UCR-ED proposes *reducing the Z-normalization cost* and *prioritizing the accumulation order*.

UCR-DTW. For DTW-based subsequence search, the UCR-DTW [5] employs a filter-and-refinement framework. It evaluates the DTW distance for a subsequence only if it survives from three lower-bounds, i.e., $LB_{Kim\mathbf{FL}}$, LB^{EQ}_{Keogh} and LB^{EC}_{Keogh}. More specifically, the lower-bounds are applied in an order starting from quick-and-dirty one to slow-and-accurate one, as shown in Figure 2. We briefly introduce these lower bounds as follows.

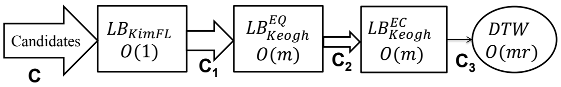


Figure 2: UCR-DTW framework

$\underline{LB_{Kim\mathbf{FL}}}$ is based on a fact that the **F**irst and the **L**ast offset must be matched in DTW.

$\underline{LB^{EQ}_{Keogh}}$ is derived from the distance between the candidate subsequence $\hat{t}_c$ and the envelop of $\hat{q}$. The envelop of $\hat{q}$ is based on the warping constraint r where $\hat{q}[i]$ can be matched with $\hat{t}_c[j]$ subject to $|j - i| \leq r$. The upper and lower envelop can be calculated as, $\hat{q}^u[i] = \max_{j=i-r}^{i+r} \hat{q}[j]$ and $\hat{q}^l[i] = \min_{j=i-r}^{i+r} \hat{q}[j]$, respectively. Accordingly, we have

$$LB^{EQ}_{Keogh}(\hat{q},\hat{t}_c) = \sqrt{\sum_{i=1}^{m} \begin{cases} (\hat{t}_c[i] - \hat{q}^u[i])^2 & \text{if } \hat{t}_c[i] > \hat{q}^u[i] \\ (\hat{t}_c[i] - \hat{q}^l[i])^2 & \text{if } \hat{t}_c[i] < \hat{q}^l[i] \\ 0 & \text{otherwise} \end{cases}} \quad (3)$$

$\underline{LB^{EC}_{Keogh}}$ is similar to LB^{EQ}_{Keogh} but the lower bound is derived from the distance between the query $\hat{q}$ and the envelop of $\hat{t}_c$. It should be noted that the optimization techniques in UCR-ED can be applied to compute LB^{EQ}_{Keogh} and LB^{EC}_{Keogh}.

3. LOWER-BOUND OPTIMIZATIONS

In this section, we propose two online lower-bounds, i.e., taking $O(1)$ and $O(\phi)$ time[1], to further improve the performance of the UCR Suite for both ED and DTW.

3.1 Fast ED-based Subsequence Search

The UCR Suite does not employ any lower-bound for the Euclidean distance (ED) [5], even though ED takes $O(m)$ time. In this section, we propose two lower bounds for ED, which can be computed in $O(1)$ and $O(\phi)$, respectively.

3.1.1 *Lower-Bound by Triangle Inequality in $O(1)$ Time*

As shown in Lemma 1, we can derive the Euclidean distance lower-bound for the running candidate t_c based on the exact distance of the last candidate (i.e., the consecutive subsequence t_{c-1}) by triangle inequality.

LEMMA 1 (LOWER-BOUND FOR ED). *For two consecutive candidate subsequences t_{c-1} and t_c in t, we have:*

$$ED(\hat{q},\hat{t}_{c-1}) - ED(\hat{t}_{c-1},\hat{t}_c) \leq ED(\hat{q},\hat{t}_c)$$

PROOF. It is trivial as triangle inequality holds for the Euclidean distance. □

We define $LB^{TRI}_{ED}(\hat{q},\hat{t}_c)$ as follows:

$$LB^{TRI}_{ED}(\hat{q},\hat{t}_c) = LB_{ED}(\hat{q},\hat{t}_{c-1}) - ED(\hat{t}_{c-1},\hat{t}_c), \quad (4)$$

where $LB_{ED}(\hat{q},\hat{t}_{c-1})$ is any lower bound which satisfies $LB_{ED}(\hat{q},\hat{t}_{c-1}) \leq ED(\hat{q},\hat{t}_{c-1})$.

With Lemma 1, we have $LB^{TRI}_{ED}(\hat{q},\hat{t}_c) \leq ED(\hat{q},\hat{t}_c)$. Suppose the $LB_{ED}(\hat{q},\hat{t}_{c-1})$ is known, we will elaborate how to compute it shortly. $LB^{TRI}_{ED}(\hat{q},\hat{t}_c)$ can be computed in $O(1)$ iff $ED(\hat{t}_{c-1},\hat{t}_c)$ can be calculated in constant time. To achieve this, it is sufficient to maintain five running sums: $S_1 = \sum_{i=1}^m t_{c-1}[i]$, $S_2 = \sum_{i=1}^m t_c[i]$, $S_3 = \sum_{i=1}^m t^2_{c-1}[i]$, $S_4 = \sum_{i=1}^m t^2_c[i]$, and $S_5 = \sum_{i=1}^m t_{c-1}[i]t_c[i]$, where these running sums can be maintained incrementally in $O(1)$ time. With these running sums, we can compute $ED(\hat{t}_{c-1},\hat{t}_c)$ in $O(1)$ as follows.

$$ED(\hat{t}_{c-1},\hat{t}_c) = \sqrt{2m(1 - \rho(t_{c-1},t_c))} \quad (5)$$

where $\rho(t_{c-1},t_c)$ is the Pearson correlation between t_{c-1} and t_c.

$$\rho(t_{c-1},t_c) = \frac{mS_5 - S_1S_2}{\sqrt{mS_3 - (S_1)^2}\sqrt{mS_4 - (S_2)^2}} \quad (6)$$

3.1.2 *Lower-Bound by PAA in $O(\phi)$ Time*

The Piecewise Aggregate Approximation (PAA) [3] is a concise representation for time series. Given a normalized subsequence $\hat{t}_c$, its PAA representation is a ϕ-dimensional vector where the k-th element is defined as follows.

$$e_{\hat{t}_c}[k] = \frac{\phi}{\ell} \sum_{x=\frac{\ell}{\phi}\cdot k}^{\frac{\ell}{\phi}(k+1)-1} \hat{t}_c[x] \quad (7)$$

The distance, $LB^{PAA}_{ED}(\hat{q},\hat{t}_c)$, between the PAA representations of $\hat{q}$ and $\hat{t}_c$ is:

$$LB^{PAA}_{ED}(\hat{q},\hat{t}_c) = \sqrt{\frac{m}{\phi}\sum_{k=0}^{\phi-1}(e_{\hat{q}}[k] - e_{\hat{t}_c}[k])^2}$$

[1] The value of parameter ϕ is much smaller than m typically, i.e., $\phi \ll m$.

According to [3], $LB_{ED}^{PAA}(\hat{q}, \hat{t}_c) \leq ED(\hat{q}, \hat{t}_c)$. It can be calculated in $O(\phi)$ time with the PAA representations of $\hat{q}, \hat{t}_c$. Our remaining challenge is to compute the PAA of a subsequence efficiently, where a straightforward solution takes $O(m)$ time. To avoid $O(m)$ time cost, we transform Eq. 7 (i.e., expanding $\hat{t}_c[x]$) into the following equation.

$$e_{\hat{t}_c}[k] = \frac{\phi}{m} \sum_{x=\frac{m}{\phi}\cdot k}^{\frac{m}{\phi}(k+1)-1} \frac{t[c+x] - \mu(t_c)}{\sigma(t_c)} \tag{8}$$

Observe that $\mu(t_c)$, $\sigma(t_c)$, $\sum t[c+x]$ can be calculated in O(1) time by $\sum t_c[i]$, $\sum t_c^2[i]$ and $\sum t[c+x-1]$, respectively. Thus we can compute the PAA of $\hat{t}_c$ in $O(\phi)$ incrementally. This optimization is also utilized in [4].

3.1.3 Putting Them Altogether

Returning to the Equation 4, we require $LB_{ED}(\hat{q}, \hat{t}_{c-1}) \leq ED(\hat{q}, \hat{t}_{c-1})$. Thus, $LB_{ED}(\hat{q}, \hat{t}_{c-1})$ can be updated to $LB_{ED}^{TRI}(\hat{q}, \hat{t}_{c-1})$, $LB_{ED}^{PAA}(\hat{q}, \hat{t}_{c-1})$ and $ED(\hat{q}, \hat{t}_{c-1})$.

Algorithm 1 shows the pseudo code for ED-based subsequence search with our two novel lower bounds. We denote LB_{ED} as the lower bound for the current candidate t_c. We apply a cheap triangle inequality bound (cf. Line 5), before applying a slightly expensive PAA bound (cf. Line 7). If t_c cannot be pruned by all lower bound testings, then we compute the distance $ED(\hat{q}, \hat{t}_c)$ by calling UCR-ED. Furthermore, the value of LB_{ED} in the current iteration will be used to derive the bound in the next iteration.

Algorithm 1 ED-based subsequence search

```
    Alg ED-search(Query $q$, Sequence $t$, Dimensionality $\phi$)
 1: $os := -1, bsf := \infty, LB_{ED} := 0$
 2: for $c := 1$ to $n - m + 1$ do
 3:     if $c > 1$ and $LB_{ED} > bsf$ then
 4:         $LB_{ED} := LB_{ED} - ED(\hat{t}_{c-1}, \hat{t}_c)$                ▷ $O(1)$
 5:     if $LB_{ED} < bsf$ then
 6:         $LB_{ED} := LB_{ED}^{PAA}(\hat{q}, \hat{t}_c)$                    ▷ $O(\phi)$
 7:         if $LB_{ED} < bsf$ then
 8:             compute $ED(\hat{q}, \hat{t}_c)$ using UCR-ED                 ▷ $O(m)$
 9:             if $ED(\hat{q}, \hat{t}_c) < bsf$ then
10:                 $os := c, bsf := ED(\hat{q}, \hat{t}_c)$
11:             $LB_{ED} := ED(\hat{q}, \hat{t}_c)$
12: return $(os, bsf)$
```

We illustrate Algorithm 1 using an example in Figure 3. Suppose $bsf = 2$ and we can safely prune t_1 by computing its lower-bound, $LB_{ED}^{PAA}(\hat{q}, \hat{t}_1) = 10$. As $ED(\hat{t}_1, \hat{t}_2)$ = 4 (i.e., using Equation. 5-6), thus we can derive $LB_{ED}(\hat{q}, \hat{t}_2) = (10 - 4) = 6$. Since $LB_{ED}(\hat{q}, \hat{t}_2)$ is larger than bsf, t_2 is safely pruned. Based on the triangle inequality, we have $ED(\hat{q}, \hat{t}_3) \geq LB_{ED}(\hat{q}, \hat{t}_2) - ED(\hat{t}_2, \hat{t}_3) = (LB_{ED}(\hat{q}, \hat{t}_1) - ED(\hat{t}_1, \hat{t}_2)) - ED(\hat{t}_2, \hat{t}_3)$. By computing $ED(\hat{t}_2, \hat{t}_3)$, we can derive $LB_{ED}(\hat{q}, \hat{t}_3) = (10 - 4) - 3 = 3 > bsf$, as a consequence, t_3 can also be pruned safely.

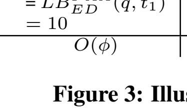
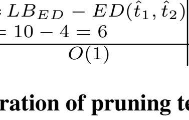
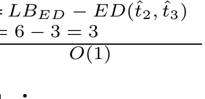
	subsequence t_1	**subsequence t_2**	**subsequence t_3**
Dist			
LB	= $LB_{ED}^{PAA}(\hat{q}, \hat{t}_1)$ $= 10$	= $LB_{ED} - ED(\hat{t}_1, \hat{t}_2)$ $= 10 - 4 = 6$	= $LB_{ED} - ED(\hat{t}_2, \hat{t}_3)$ $= 6 - 3 = 3$
Time	$O(\phi)$	$O(1)$	$O(1)$

Figure 3: Illustration of pruning techniques

3.2 Fast DTW-based Subsequence Search

The proposed lower-bounds in Section 3.1.1 can be adapted to $LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_c)$. In this section, we discuss two lower-bounds for LB_{Keogh}^{EQ} whose computation cost are $O(1)$ and $O(\phi)$, respectively.

3.2.1 Lower-Bound by Triangle Inequality in $O(1)$ Time

As shown in Lemma 2, we can derive the lower-bound of $LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_c)$ based on the $LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_{c-1})$ by triangle inequality.

LEMMA 2 (LOWER-BOUND FOR LB_{Keogh}^{EQ}). *Let t_{c-1} and t_c be consecutive candidates in t. We have:*

$$LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_{c-1}) - ED(\hat{t}_{c-1}, \hat{t}_c) \leq LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_c)$$

PROOF. For LB_{Keogh}^{EQ}, there exists a sequence a that is enclosed by the envelop sequences $\hat{q}^u$ and $\hat{q}^l$ (i.e., $\hat{q}^l[i] \leq a[i] \leq \hat{q}^u[i]$), such that $LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_c) = ED(a, \hat{t}_c)$. With Lemma 3, we have $LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_{c-1}) \leq ED(a, \hat{t}_{c-1})$.

$$LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_{c-1}) \leq ED(a, \hat{t}_{c-1})$$
$$\iff LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_{c-1}) - ED(\hat{t}_{c-1}, \hat{t}_c)$$
$$\leq ED(a, \hat{t}_{c-1}) - ED(\hat{t}_{c-1}, \hat{t}_c)$$

Applying Lemma 1: $ED(a, \hat{t}_{c-1}) - ED(\hat{t}_{c-1}, \hat{t}_c) \leq ED(a, \hat{t}_c)$

$$\iff LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_{c-1}) - ED(\hat{t}_{c-1}, \hat{t}_c) \leq ED(a, \hat{t}_c)$$
$$\iff LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_{c-1}) - ED(\hat{t}_{c-1}, \hat{t}_c) \leq LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_c)$$

□

LEMMA 3. *$LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_c) \leq ED(a, \hat{t}_c)$, where sequence a is enclosed by sequences $\hat{q}^u$ and $\hat{q}^l$, i.e., $\hat{q}^l[i] \leq a[i] \leq \hat{q}^u[i]$.*

PROOF. Since $\hat{q}^l[i] \leq a[i] \leq \hat{q}^u[i]$, for case (i) $\hat{t}_c[i] > \hat{q}^u[i]$, we have $(\hat{t}_c[i] - \hat{q}^u[i])^2 \leq (\hat{t}_c[i] - a[i])^2$. Similarly, for case (ii) $\hat{t}_c[i] < \hat{q}^l[i]$, we have $(\hat{t}_c[i] - \hat{q}^l[i])^2 \leq (\hat{t}_c[i] - a[i])^2$ as $a[i] \geq \hat{q}^l[i]$. Finally, $0 \leq (\hat{t}_c[i] - a[i])^2$ always hold for case (iii) $\hat{q}^l[i] \leq \hat{t}_c[i] \leq \hat{q}_u[i]$. □

Similarly, we define $LB_{DTW}^{TRI}(\hat{q}, \hat{t}_c)$ as follows:

$$LB_{DTW}^{TRI}(\hat{q}, \hat{t}_c) = LB_{DTW}(\hat{q}, \hat{t}_{c-1}) - ED(\hat{t}_{c-1}, \hat{t}_c), \tag{9}$$

where $LB_{DTW}(\hat{q}, \hat{t}_{c-1})$ is any lower bound with $LB_{DTW}(\hat{q}, \hat{t}_{c-1}) \leq LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_{c-1})$. With Lemma 2, we have $LB_{DTW}^{TRI}(\hat{q}, \hat{t}_c) \leq LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_c)$.

3.2.2 Lower-Bound by PAA in $O(\phi)$ Time

In section 3.1.2, we propose a technique to construct the PAA representations for subsequences incrementally in $O(\phi)$ time. Given the PAA representations, we can derive $LB_{DTW}^{PAA}(\hat{q}, \hat{t}_c)$ as follows.

$$LB_{DTW}^{PAA}(\hat{q}, \hat{t}_c) = \sqrt{\frac{m}{\phi} \sum_{x=0}^{\phi-1} \begin{cases} (e_{\hat{t}_c}[x] - \hat{U}[x])^2 & e_{\hat{t}_c}[x] > \hat{U}[x] \\ (e_{\hat{t}_c}[x] - \hat{L}[x])^2 & e_{\hat{t}_c}[x] < \hat{L}[x] \\ 0 & \text{otherwise} \end{cases}}$$

where $\hat{U}$ and $\hat{L}$ are the PAA representation of the upper and lower envelopes of $\hat{q}$, respectively. In addition, its building cost can be neglected as it is only computed once at the beginning of the search.

According to [2], we have $LB_{DTW}^{PAA}(\hat{q}, \hat{t}_c) \leq LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_c)$. Returning to Lemma 2, we require $LB_{DTW}(\hat{q}, \hat{t}_{c-1}) \leq LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_{c-1})$. Thus, $LB_{DTW}(\hat{q}, \hat{t}_{c-1})$ can be set to $LB_{DTW}^{TRI}(\hat{q}, \hat{t}_{c-1})$, $LB_{DTW}^{PAA}(\hat{q}, \hat{t}_{c-1})$ or $LB_{Keogh}^{EQ}(\hat{q}, \hat{t}_{c-1})$.

3.2.3 *Putting Them Altogether*

Figure 4 illustrates the complete framework of our DTW-based subsequence search. We first evaluate $LB_{Kim\mathbf{FL}}$ as in the UCR Suite. We proceed to examine the lower-bound by triangle inequality if the candidates cannot be pruned by $LB_{Kim\mathbf{FL}}$. As a remark, $LB_{DTW}^{TRI}(\hat{q}, \hat{t}_c)$ is computed only when $LB_{DTW}(\hat{q}, \hat{t}_{c-1}) > bsf$. The surviving candidates are then evaluated by $LB_{DTW}^{PAA}(\hat{q}, \hat{t}_c)$, its cost is $O(\phi)$, before computing expensive LB_{Keogh}^{EQ} and LB_{Keogh}^{EC} (i.e., $O(m)$). Finally, we call $DTW(\hat{q}, \hat{t}_c)$ if $\hat{t}_c$ is not pruned by any above lower bounds.

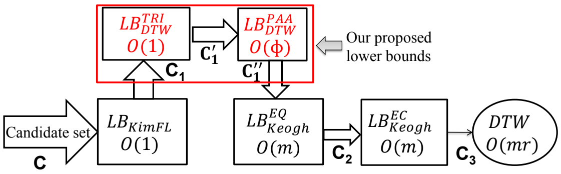


Figure 4: DTW-based subsequence search

4. EXPERIMENT

In this section, we compare our proposed techniques (denoted as FAST-ED and FAST-DTW), with the UCR Suite (denoted as UCR-ED and UCR-DTW).

Platform setting: All methods are implemented in C++. We evaluate the performance on a machine running 64-bit Windows 10 with a 3.16GHz Intel(R) Core(TM) Duo CPU E8500, 8 GB RAM.

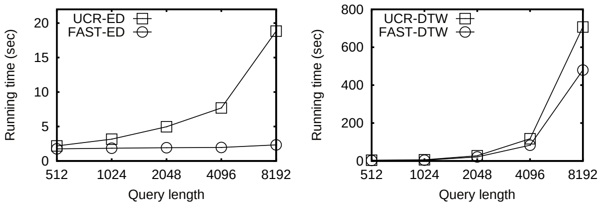
Dataset: To evaluate the efficiency and scalability of our proposed techniques, we use the random walk model as in [4, 5] to generate both the query and the target time series. For each experimental setting, we run each method by 10 query sequences, and report the average execution time.

Result: We first test the performance of FAST-based subsequence search methods by varying ϕ from 8 to 40, where query length is 1024 and data length is 2 millions. The maximum standard deviation among these execution time is less than 3% in both FAST-ED and FAST-DTW. Thus, we choose $\phi = 24$ in the following experiments.

Table 1 compares the pruning ratio of our proposed lower-bounds with existing lower-bounds in the UCR Suite (in gray color). The UCR-ED does not provide any lower-bound function whereas our proposed lower-bounds can prune at least 99% of candidates. For example, it can prune 99.8% candidates when query length equals to 512 (i.e., among them, 14.6% candidates are pruned by LB_{ED}^{PAA} and 85.2% candidates are pruned by LB_{ED}^{TRI}). For subsequence search under DTW, our proposed lower-bounds are applied after LB_{KimFL} and before LB_{Keogh}^{EQ} (cf. Figure 4). Thus, the proposed lower-bounds can absorb most the pruning ability of LB_{Keogh}^{EQ}, but not other existing lower-bounds like $LB_{KimFL}, LB_{Keogh}^{EC}$. As shown in Table 1, our proposed lower-bounds can nearly approach the pruning ability of LB_{Keogh}^{EQ} by using less CPU time. Recall that our proposed bounds take $O(1)$ and $O(\phi)$ time, whereas LB_{Keogh}^{EQ} takes $O(m)$ time. Figure 5 shows the execution time of the FAST-based and the UCR-based subsequence search, by varying the query length, where data length is 2 million. The FAST-based search can be up to 11 and 3 times faster than the UCR-based search on ED and DTW respectively. Note that the performance gap between the FAST-based and the UCR-based widens when the query length increases.

Table 1: Pruning ratios, on RW

Query Length	512	1024	2048	4096	8192
UCR-ED: N/A	-				
LB_{ED}^{PAA}	14.6%	11.8%	9.4%	7.1%	5.4%
LB_{ED}^{TRI}	85.2%	88.0%	90.3%	92.6%	94.2%
Total of proposed LBs	**99.8%**	**99.8%**	**99.7%**	**98.7%**	**99.6%**
UCR-DTW: LB_{Keogh}^{EQ}	44.0%	56.9%	80.9%	74.1%	95.0%
LB_{DTW}^{PAA}	12.0%	12.2%	12.1%	9.1%	7.9%
LB_{DTW}^{TRI}	30.8%	43.2%	67.2%	63.7%	86.4%
Total of proposed LBs	**42.8%**	**55.4%**	**79.3%**	**72.8%**	**94.1%**



(a) for ED (b) for DTW

Figure 5: Response time, on RW

5. CONCLUSION

In this paper, we propose two novel lower-bounds to speedup the subsequence search over time series data. These lower-bounds are based on the triangle inequality and PAA representations, respectively, and can be easily integrated with the UCR Suite to further improve its performance.

Acknowledgement

This work was supported by grant GRF 152043/15E from the Hong Kong RGC, grant MYRG105- FST13- GZG, MYRG2014-00106-FST, MYRG2015-00070-FST and MYRG2016-00182-FST from UMAC Research Committee, grant FDCT/116/2013/A3 and FDCT/007/2016/AFJ from Macau FDCT, grant NSFC 61502548 from National Natural Science Foundation of China. We thank Jiahao Zhang for helping with the experiments.

6. REFERENCES

[1] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, 1994.

[2] E. Keogh and C. A. Ratanamahatana. Exact indexing of dynamic time warping. *Knowl. Inf. Syst.*, 7(3):358–386, 2005.

[3] E. J. Keogh, K. Chakrabarti, M. J. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowl. Inf. Syst.*, 3(3):263–286, 2001.

[4] Y. Li, L. H. U, M. L. Yiu, and Z. Gong. Quick-motif: An efficient and scalable framework for exact motif discovery. In *ICDE*, pages 579–590, 2015.

[5] T. Rakthanmanon, B. J. L. Campana, A. Mueen, G. E. A. P. A. Batista, M. B. Westover, Q. Zhu, J. Zakaria, and E. J. Keogh. Addressing big data time series: Mining trillions of time series subsequences under dynamic time warping. *TKDD*, 7(3):10, 2013.

Progressive Recovery of Correlated Failures in Distributed Stream Processing Engines

Li Su
University of Southern Denmark
lsu@imada.sdu.dk

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

ABSTRACT

Correlated failures in large-scale clusters have significant effects on systems' availability, especially for streaming data applications that run continuously and require low processing latency. Most state-of-the-art distributed stream processing engines (DSPEs) adopt a blocking recovery paradigm, which, upon correlated failure, would block the progress of recovery until sufficient new resources for recovery are available. As the arrival of new resources is usually progressive, a blocking paradigm fails to minimize the recovery latency. To address this problem, we propose a progressive and query-centric recovery paradigm where the recovery of the failed operators would be carefully scheduled to progressively recover the outputs of queries as early as possible based on the current availability of resources. In this work, we propose and implement a fault-tolerance framework which supports progressive recovery after correlated failures with minimum overhead during the system's normal execution. We also formulate the new problem of recovery scheduling under correlated failures and design effective algorithms to optimize the recovery latency. The proposed methods are implemented on Apache Storm and preliminary experiments are conducted to verify their validity.

1. INTRODUCTION

Fault tolerance is critical to Distributed Stream Processing Engines (DSPEs), such as Apache Storm [14] and Spark Streaming [3], mainly due to the long running time and the low latency requirement of streaming data applications. Previous researches in this area are mainly focused on individual and independent node failures and ignore correlated failures [7, 8], where a number of nodes fail within a short interval. Correlated failures can be caused by failures of shared hardwares, such as switches, routers, and power facilities, or by software problems, such as bad software patches applied across a number of nodes. Although large-scale correlated failures occur less frequently than independent ones, they have significant effects on a system's availability [7].

Correlated failures exhibit characteristics that are very different from independent failures. First of all, correlated failures would incur the unavailability of a large amount of resources. One cannot assume an instant availability of sufficient resources to recover the continuous queries from such failures. Repairing the failed nodes or acquiring additional resources would take a significant amount of time. For example, it may involve solving the software or hardware problems, restarting the failed nodes, and adding them back to the DSPE. Even if the DSPE is running on a cloud environment and virtual resources can be easily allocated to replace the failed nodes, negotiating and acquiring a large amount of new resources would still incur a latency non-negligible for streaming data applications. More importantly, the recovered or newly allocated nodes would probably not become available simultaneously, but rather one after another with noticeable time gaps between them. In other words, the current assumption that all the resources needed for recovery are available at the same time cannot be held.

Most existing DSPEs, such as Flink [1] and Storm [14], adopt a blocking recovery approach in the sense that the recovery of all the parallel operator partitions would be blocked until sufficient new resources are acquired. However, due to the gradual availability of resources in the recovery of correlated failures, such a blocking approach fails to minimize the recovery latency. It is much more desirable to adopt a progressive recovery approach, where the operator partitions can be recovered progressively upon the availability of new resources. Furthermore, the existing systems also adopt an operator-centric paradigm in the scheduling of the recovery, where the operator partitions are scheduled for recovery individually in a topological order. Note that the accurate outputs of a query can only be generated if and only if all the operator partitions of this query are executing normally, this operator-centric paradigm fails to minimize the latency of recovering the producing of query outputs. To address the insufficiency of the existing approaches, we propose a progressive and query-centric recovery paradigm where the recovery of the failed operator partitions would be progressively scheduled to recover the outputs of queries as early as possible based on the current availability of resources. More specifically, if correlated failure happens, we gradually increase the number of recovered queries following the arrival pace of the restarted or the newly acquired nodes. Furthermore, unlike the operator-centric paradigm, our query-centric paradigm attempts to schedule the recovery of the failed partitions to produce the output of a query as soon as possible. This new paradigm would provide not only a shorter recovery latency and earlier query results, but also a more responsive and smoother transition from a failed state to a fully recovered one.

In summary, we propose a fault-tolerance framework that can support progressive recovery during a correlated failure, which imposes minimum overhead during the system's normal execution. We also formulate the new problem of query-centric recovery scheduling under correlated failures, which is an NP-hard problem. To provide a solution for a large-scale job topology, we propose an ef-

Series ISSN: 2367-2005

10.5441/002/edbt.2017.59

ficient and effective approximate algorithm. We implement the recovery framework and the scheduling algorithm on top of Apache Storm, a popular and mature open-source DSPE and conduct multiple sets of experiments on Amazon EC2 to validate the effects of progressive recovery.

2. RELATED WORK

Fault tolerance for DSPEs can be generally categorized into two types [9]: passive approaches and active approaches. Passive techniques include checkpoint [1], upstream buffer [6, 12] and source replay [14, 1, 2]. Active approaches [4, 5, 13, 12] employ hot-standby replicas to achieve faster failure recovery with higher resource consumption. The mainstream DSPEs, such as Samza [2], Flink [1] and Storm [14] adopt source replay and checkpointing techniques. Our checkpointing scheme is similar to the one used in [1]. Both the works in [6, 12] combine checkpointing and upstream buffer to achieve fault tolerance as we do in this work, while missing the optimization for recovery scheduling makes them not suitable for progressively recovering large-scale correlated failures. [13] presents a framework to combine both active and passive techniques to maximize the accuracy of the fast tentative query outputs in correlated failure. Different from [13], which mainly focuses on optimizing resource assignment to improve the quality of tentative outputs, our approach focuses on progressive recovery that minimizes the latency of completely recovering correlated failures, which is orthogonal to the problem studied in [13].

3. PRELIMINARIES

As in most of the mainstream DSPEs, such as Storm [14] and Samza [2], we model a data tuple as a $\{key, value\}$ pair, where the default format of the key is string and the value is a blob that is opaque to the system. The execution plan of a query consists of multiple operators, each of which contains a user-defined function and can subscribe the output streams of other operators. An operator can be parallelized into multiple operator partitions that have identical computation logic defined by the user-defined function of the operator. Each input stream of an operator is split into a set of key groupings based on their keys. A union of the same key grouping from each of the input streams of an operator would form the complete input of an operator partition, which is also referred to as partition for simplicity throughout this work.

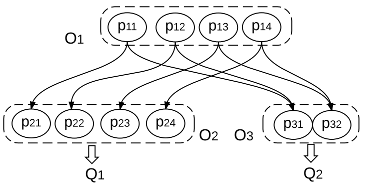


Figure 1: An example topology which consists of two queries Q_1 and Q_2, whose operator sets are $\{O_1, O_2\}$ and $\{O_1, O_3\}$, respectively.

By denoting the operator partitions as vertex and data streams between the operator partitions as directed edges, the execution plan of a query can be abstracted as a directed acyclic graph (DAG). Figure 1 depicts an example DAG. The computation states, input and output buffers for each partition are maintained separately from each other. The output stream of an operator can be shared by the execution plans of multiple queries. Therefore, the DAGs of queries are connected by the shared vertex. We refer to the topology that is composed by all the queries which are concurrently running within the DSPE as the global topology. A user-specified priority, which is denoted as a numerical value (set as 1 by default), is assigned to each query within the topology

4. FAULT TOLERANCE

In this section, we present the fault-tolerance framework that supports progressive recovery and some implementation details.

Checkpointing We use punctuations to trigger checkpointing in partitions and synchronize the progress of checkpoints. Punctuations are generated periodically and inserted into the source streams in a broadcasting fashion. On receiving the punctuations with the same sequence number from all the input streams, a partition starts the process of checkpointing and then broadcasts this punctuation to its downstream neighboring partitions. As the punctuations are not arriving simultaneously, data items arrive after the punctuations must be buffered before the checkpoint is done. Assuming that the last checkpoint of a partition is triggered by punctuation P_{k-1}, tuples from S_i which are received after P_k will be stored in the input buffer. After receiving P_k from all the input streams, the partition generates a checkpoint that stores its computation state and then acknowledges the coordinator. The coordinator tracks the checkpointing progress of the whole topology. Once the coordinator is acknowledged that all the partitions have completed checkpointing for punctuation P_k, it knows that a global synchronized checkpoint of the entire topology for P_k, denoted by $cp(P_k)$, is generated.

Adaptive Buffering. Source buffering is a widely adopted fault-tolerance technique in DSPEs. With source buffering, the system buffers the source data of which the processing state have not been included in the latest global checkpoint. In other words, when a global checkpoint of the entire topology is completely made, we can trim the source buffers by removing those source data whose processing are already reflected in the global checkpoint. It is important to note that the buffers for failure recovery differ from the buffers used in data transfer. The latter can be easily trimmed whenever the data are transferred to the downstream nodes. Due to its simplicity and low overhead, the source buffering approach is widely adopted in most existing operational DSPEs, including Storm [14], Flink [1] and Samza [2]. Upstream buffering is another bufffering technique that requires each partition to buffer its own output until a global checkpoint is made. Due to its high overhead during normal execution, this approach is not used in most mainstream DSPEs.

However, source buffering cannot support progressive recovery, because whenever we need to recover the state of a partition, we have to replay the buffered data from the sources till the current partition. Only recovering a part of the failed partitions makes little sense because the recovery of any remaining one would require to redo the whole recovery again. This means the recovery progress should be blocked until there are sufficient resources to recover all the failed partitions. To solve the above problem, we adopt an approach, called adaptive buffering, which would only incur overhead during failure recovery. With adaptive buffering, we only buffer at the sources during normal execution. Once a burst of multiple node failures is detected within a time window, all the partitions except for the sinks would buffer their outputs to support progressive recovery. These output buffers are turned off when a new global checkpoint is completely created, which indicates the correlated failure is completely recovered.

Figure 2 presents an example of adaptive buffering. Before failure is detected, only the partition in the source operator (i.e., p_1), has output buffer. When partitions p_3, p_4, and p_5 are detected to be failed, at timestamp ts_1, p_3 and p_4 are restarted and the output buffer is turned on in partition p_2 and p_3. At ts_2, after p_5 is restarted, it will first process the output buffer of partition p_3. After all the partitions are recovered, output buffer is turned off in the

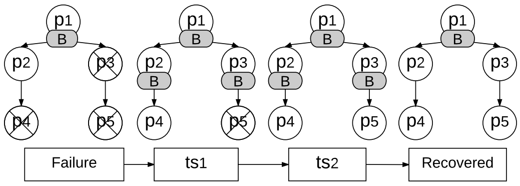


Figure 2: An example of adaptive buffering.

non-source partitions, only the output buffer in p_1 is preserved.

Progressive Recovery. Once failure is detected, assuming that P_k is the punctuation of the latest successful global checkpoint, the failed partitions are restarted and the states of the whole topology are restored or rollbacked with checkpoint $cp(P_k)$. The system would switch to the progressive recovery mode if the total number of failed nodes is higher than a threshold within a specific time window, otherwise it would simply use the blocking recovery method. With adaptive buffering, the output buffers in all the partitions are now turned on and the input data with a greater sequence number than P_k will be replayed from the sources. These output buffers could be used to resume the progress of the failed partitions that are recovered when new recovery resources arrive.

Note that node failures may not occur simultaneously during a correlated failure. In other words, it is possible that additional failures could be detected before the current recovery is completed. With the adaptive upstream buffers, instead of rolling back the states of the whole topology to $cp(P_k)$ again, we only restore the states of the newly failed partitions with $cp(P_k)$ and replay the data buffered in their upstream neighbors. However, as the progress of the newly restored partitions fall behind their downstream neighbors that have been recovered, the downstream may receive duplicated tuples and therefore have to perform duplicate elimination to guarantee exactly-once processing.

However, for a partition p_i with multiple input streams, as the tuples from different upstream partitions may arrive in different orders, p_i may produce outputs in different orders across different replays. To solve this problem, we enforce Order-Preserved processing during recovery to ensure that p_i processes its input in an identical order across different replays. The order-preserved processing is turned on in the beginning of the recovery. The source-buffered data would be divided into mini-batches and each partition attaches a local sequence number that increases monotonically to each of its output tuples. For a partition p_i, tuples within the same batch are stored in its input buffer. When it receives all the data from a batch from all its inputs, p_i starts processing these data from each input stream in a predefined round-robin order. In this way, the order of the output data are guaranteed to be identical across multiple replays. With order-preserved processing, the downstream of p_i can skip duplicate tuples by checking the sequence numbers of tuples from p_i. After the recovery is completed, the order-preserved processing will be turned off together with adaptive buffering.

Implementation. We implement our system on Enorm [11], which is a distributed stream processing system built on Apache Storm [14]. In our system, a special bolt, called control bolt, is automatically generated and appended to the user-submitted job topology. The responsibilities of the control bolt include collecting workload statistics and handling node failure. The fault tolerance coordinator in the control bolt detects node failures by checking their heartbeats in ZooKeeper. Upon a failure is detected, the coordinator calls the optimization algorithm presented in Section 5 to schedule failure recovery following the pace of acquiring new resources. The control bolt is stateless, if failed, it will be restarted by Nimbus in Storm on another node and the interrupted recovery scheduling will be resumed.

5. OPTIMIZING RECOVERY PLAN

In this section, we define the problem of optimizing the recovery scheduling and present an outline of our optimization algorithm. Given a global topology T, we denote the resource consumption of operator O_i in T as C_i, the parallelization degree of O_i as m_i and the resource consumption of p_{ij}, the jth partition of O_i, as c_{ij}. We have $C_i = \sum_{j=1}^{m_i} c_{ij}$. Queries can be assigned with priorities according to their importance and Q_i's priority is denoted by prt_i.

If the amount of available resources is not enough to recover all the failed partitions of a correlated failure, we have to select a subset of the failed partitions for recovery. Whenever a set of new nodes are available, a set of failed partitions will be scheduled for recovery, which is referred to as a partial recovery plan. A failed query is called recovered if and only if all of its failed partitions are recovered. We present a formal definition for the problem of optimizing recovery plan as follows:

RECOVERY PLAN OPTIMIZATION*: For a global topology T, a set of failed queries QS, and the amount of computation resources R available for failure recovery, choose a subset of the failed operator partitions for recovery such that the sum of the priorities of the recovered queries is maximized.*

The RECOVERY PLAN OPTIMIZATION problem is NP-hard, as it can be reduced from the Set Union Knapsack problem, which has been proved to be NP-hard [10]

Considering that operators can be shared by multiple queries, it is natural to prioritize recovering the queries whose operators are shared by more queries. Furthermore, as the failed queries have various recovery costs and priorities, we should consider the profit that can be achieved by using per unit of resource while generating the recovery plan. Taking the above two factors into consideration, we define *Profit Density*, referred to as PD_i, of query Q_i and use it to rank the recovery priorities of the failed queries. PD_i is calculated as follows:

$$PD_i = \frac{prt_i}{\sum^{O_k \in Q_j} \frac{C_k}{f_k}}$$

In the above equation, C_k is the cost of recovering the failed partitions in operator O_k, f_k is the frequency that O_k is shared by the other failed queries. The approximate optimization algorithm starts by calculating the profit density of each failed query. The failed queries are put into a list and sorted in descending order according to their profit density. Next, the list is traversed from the beginning to find the query, Q_i, whose recovery cost is smaller than the amount of currently available resources. The failed partitions belonging to Q_i will be put into the recovery plan. The profit density of the other failed queries will be updated and the list of failed queries are re-sorted. The above loop continues until the resource constraint is reached. The time complexity of this algorithm is $O\left(M^2 \cdot logM\right)$, where M is the number of the failed queries.

6. EVALUATION

All the experiments are conducted on Amazon EC2 using the m3.large instance. We use a real data set consisting of 569,382 tweets crawled from Twitter, which are repeatedly emitted in order into the source operator to emulate a long-standing application.

To explore the time of attaching new nodes to a cluster on a cloud platform, e.g., Amazon EC2, we conduct experiments to record the time interval between when the instance acquiring is started and when the newly attached node is ready to host processing task. We collect in total 180 samples and present their distribution in Fig-

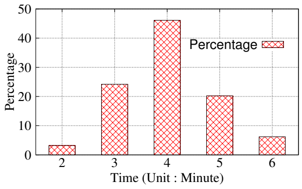


Figure 3: Distribution of time cost of attaching new nodes to a deployed cluster.

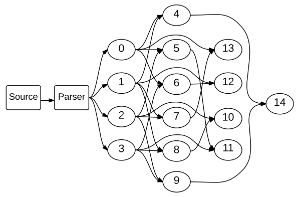
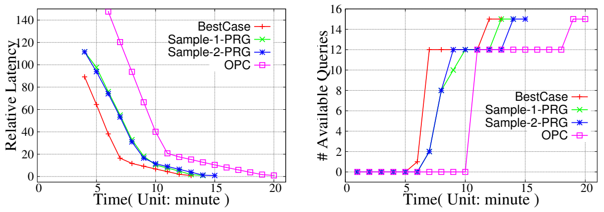


Figure 4: Topology used in the experiment of progressive recovery.

ure 3. One can see that, even on the cloud platform, the newly attached nodes are not arriving simultaneously. The time to attach a new node varies from 2 minutes to 6 minute. This result consolidates our motivation for progressive recovery.



(a) Relative Latency

(b) Available Queries

Figure 5: Average relative latency of recovered queries and the number of available queries after correlated failure.

Figure 4 shows the structure of the job topology used in the recovery experiments. There are 15 queries in this topology. The sink operator of query Q_i is denoted as O_i. The parallelization degree is set as 1 for the *Source* and 5 for the other operators. The *Source* operator emits tweets in the rate of 1000 tuples per second. On receiving a tweet, the *Parser* emits a tuple for each hashtag within the tweet. Operator O_1, O_2, O_3, and O_4 conduct sliding-window aggregates, which count the hashtag frequency with various window settings and output the updates of the window instances. Operator O_i, $4 \le i \le 14$, maintains the states of the sliding-window aggregates it subscribes.

End-to-end processing latency is a critical performance metric for most streaming data applications. As recovering a large-scale correlated failure would inevitably incur significant increment on processing latency, we propose two metrics that are relevant with processing latency to measure the effectiveness of the compared recovery schemes. Assuming that the latencies of queries before the failure are stable, we propose **Relative Latency** that measures the difference of a query's latency before and after failure. Denoting l_s as a query's latency before failure and l_r as that after failure, its relative latency, RL, is calculated as $\frac{l_r}{l_s}$. Therefore, after query Q_i is recovered, RL_i would gradually approximate 1. Within a time interval Δ_T, if the average RL_i of Q_i is smaller than Θ, e.g., $\Theta = 1.2$ in this set of experiments, Q_i is considered as an **Available Query**, which means it has recovered to a normal state. The cluster initially consists of 10 nodes, and we manually kill the 8 nodes where the sink operators of the 15 queries are deployed to inject a correlated failure, and then 8 new nodes are acquired and attached to the cluster to perform recovery.

Figure 5(a) and Figure 5(b) present the relative latency of the recovered queries and the number of available queries using different recovery paradigms. In both figures, *BestCase* denotes the case where all the new nodes become available simultaneously after 3 minutes and the recovery of all the failed partitions are started immediately after that. *Sample-1-PRG* and *Sample-2-PRG* are two different runs using progressive recovery and *OPC* represents the blocking operator-centric recovery.

As one can see in Figure 5, *BestCase* outperforms the others in both the relative recovery latency and the number of available queries, this is because all the failed partitions are recovered only 3 minutes after the failure. On the contrary, *OPC* has the worst recovery performance as its recovery is started after all the new nodes are ready, which results in that *OPC* has more input tuples buffered than the others before the recovery is started. The relative latency of *OPC* is nearly 50% higher than that of *BestCase* at the beginning of the recovery, and it also takes more time for the average relative latency of *OPC* to return to the stable level than *BestCase*. The relative recovery latency and the number of available queries with progressive recovery are between those of BestCase and *OPC*, as the failed partitions are gradually recovered following the pace of resource acquiring. This experiment shows that, compared to the blocking and operator-centric recovery, adopting progressive recovery brings better latency and less time for the failed queries to become available.

7. CONCLUSION

In this work, we present a query-centric progressive recovery framework to improve the efficiency of recovering correlated failure in DSPEs. Following the arriving pace of the newly acquired resources after a correlated failure, failed partitions are scheduled to be progressively recovered such that the outputs of failed queries can be generated as early as possible. We present an effective approximate algorithm to optimize the recovery plan. Experimental results show that, compared to the paradigm of blocking operator-centric recovery, our approach exhibits significant advantages while recovering correlated failures.

8. REFERENCES

[1] http://flink.apache.org/.

[2] http://samza.apache.org/.

[3] http://spark.apache.org/streaming/.

[4] M. Balazinska et al. Fault-tolerance in the borealis distributed stream processing system. SIGMOD '2005.

[5] P. Bellavista et al. Adaptive fault-tolerance for dynamic resource provisioning in distributed stream processing systems. EDBT'2014.

[6] C. Fernandez et al. Integrating scale out and fault tolerance in stream processing using operator state management. SIGMOD '2013.

[7] D. Ford et al. Availability in globally distributed storage systems. OSDI'2010.

[8] T. Heath et al. Improving cluster availability using workstation validation. SIGMETRICS '2002.

[9] J.-H. Hwang et al. High-availability algorithms for distributed stream processing. ICDE '2005.

[10] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer-Verlag Berlin Heidelberg, 2004.

[11] K. G. S. Madsen and Y. Zhou. Dynamic resource management in a massively parallel stream processing engine. CIKM '2015.

[12] A. Martin, A. Brito, and C. Fetzer. Scalable and elastic realtime click stream analysis using streammine3g. DEBS '2014.

[13] L. Su and Y. Zhou. Tolerating correlated failures in massively parallel stream processing engines. ICDE '2016.

[14] A. Toshniwal, S. Taneja, et al. Storm@twitter. SIGMOD '2014.

Optimal Obstructed Sequenced Route Queries in Spatial Databases

Anika Anwar[1], Tanzima Hashem[2]
[1,2]Department of CSE, Bangladesh University of Engineering and Technology, Bangladesh
{[1]anika.anwar@yahoo.com } {[2]tanzimahasem@cse.buet.ac.bd}

ABSTRACT

We introduce optimal obstructed sequenced route (OOSR) queries, a novel query type in spatial databases. For a given source and destination locations and the sequence of required types of points of interests (POIs) (e.g., first an ATM booth then a restaurant), an OOSR query returns the locations of POIs, one from every required type, that together minimize the obstructed trip distance (OTD) from the source to the destination via the POIs. A pedestrian's walking path is obstructed by the presence of obstacles like a river, a fence or a private property, and an obstructed distance is measured as the length of the shortest path between two locations by avoiding the obstacles. We develop the first solution to address OOSR queries. We exploit elliptical properties and develop a novel OTD computation technique that does not retrieve the same obstacles multiple times, reuses the already computed obstructed distances, and minimizes the retrieval of the extra obstacles. We propose efficient algorithms to evaluate OOSR queries with reduced IO and query processing overhead. We perform experiments using a real dataset and show a comparative analysis between OOSR algorithms.

1. INTRODUCTION

The widespread usage of location aware mobile devices has expedited the proliferation of location-based services in recent years. Researchers have proposed variant of location-based queries [1, 4, 5, 6] to assist users in planning trips in an optimized manner. In this paper, we introduce a new variant of trip planning query, an optimal obstructed sequenced route (OOSR) query that allows pedestrians to plan trips with the minimum travel distance in presence of obstacles like a river, a fence or a private property in the space. For example, a tourist walking from an attraction to the hotel may want to withdraw money from an ATM booth and then have dinner at a restaurant, or a pedestrian in the city may want to buy a medicine from a pharmacy and then visit a shopping mall before going to the bus station. An OOSR query returns the location of a point of interest (POIs) for every required type (e.g., an ATM booth or a restaurant) that together minimize the obstructed trip distance (OTD) from a user's source to the destination via the POIs. We propose the first solution for OOSR queries.

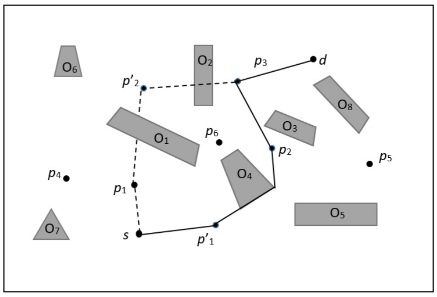


Figure 1: An example of an OOSR query

Optimal sequenced route (OSR) queries have been addressed in the unobstructed space that do not consider the presence of obstacles and cannot facilitate trip planning for pedestrians. Figure 1 shows that POIs p_1, $p_2{}'$, and p_3 minimize the trip distance if obstacles are not considered. On the other hand, the answer changes for an OOSR query as in reality pedestrians cannot cross the interior of obstacles and POIs $p_1{}'$, p_2, and p_3 minimize the OTD.

The efficiency of an OOSR algorithm depends on the OTD computation technique and the number of POIs explored for finding the optimal answer. The smaller the number of POIs retrieved from the database while searching for the optimal query answer, the more efficient the algorithm is. More importantly, the smaller number of POIs reduces the number of OTD computations. In summary, the contributions of our paper are summarized as follows:

- We introduce and formulate OOSR queries. To the best of our knowledge, we first address the OOSR query.
- We develop a novel OTD computation technique that (i) does not retrieve the same obstacles multiple times, (ii) reuses the already computed obstructed distances, and (iii) minimizes the retrieval of the extra obstacles.
- We combine the Euclidean lower bound and elliptical properties to prune POIs that cannot be part of the optimal answer, and develop efficient algorithms for processing OOSR queries with reduced IO and processing overheads.
- We compare the efficiency of our algorithms through extensive experiments using real datasets.

2. PROBLEM FORMULATION

An OOSR query is formally defined as follows:

Definition: Optimal Obstructed Sequenced Route (OOSR) Queries. Given a set of POIs P and a set of obstacles O in a 2-dimensional space, a source location s, a destination location d, and a set of m sequenced POI types $T = \{t_1, t_2, \ldots, t_m\}$, an OOSR query returns $A = \{p_{t_1}, p_{t_2}, \ldots, p_{t_m}\}$, a POI from every required type, where A minimizes the obstructed trip distance (OTD).

An obstacle o_i is a polygon in a 2-dimensional space and an obstructed space does not allow a pedestrian to cross the obsta-

Series ISSN: 2367-2005

10.5441/002/edbt.2017.60

cle like a river, a fence or a private property. An obstructed distance $dist_o(.,.)$ between two locations is measured as the length of the shortest path between two locations by avoiding the obstacles. An OTD $Tdist_o(s,d,A)$ is measured as $dist_o(s,p_{t_1}) + \sum_{i=1}{}^{m-1} dist_o(p_{t_i},p_{t_{i+1}}) + dist_o(p_{t_m},d)$.

The set of POIs P and the set of obstacles O are indexed using two separate R-trees, POI RTree and Obstacle RTree, respectively, in the database of a location-based service provider (LSP). When a user requests an OOSR query to the LSP, the LSP evaluates the OOSR query and returns the answer to the user.

3. RELATED WORK

Trip planning queries [4] and variants [5, 6] have been extensively studied in the literature. A trip planning algorithm was introduced in [4], where a user can visit POIs in any sequence that minimizes the trip distance. In [6], the authors first addressed an optimal sequenced route (OSR) query that allows users to specify the sequence of visiting POI types. In [7], the authors focus on protecting location privacy of users while evaluating an optimal trips. However, none of the above approaches consider the presence of obstacles while evaluating the queries.

Researchers have recently focused on developing algorithms for processing variant queries in the obstructed space. In [3, 10], the authors proposed algorithms for processing nearest neighbor queries in the obstructed space. The approaches in [8, 9] evaluate group nearest neighbor queries in the presence of obstacles, whereas the focus of [2] is on obstructed reverse nearest neighbor queries.

Algorithm 1 CompOTD($s,d,pTrip$)

```
Input: $s$, $d$, and a set of POIs $pTrip = \{p_{t_1}, p_{t_2}, \ldots, p_{t_m}\}$
Output: The obstructed trip distance $Tdist_O(s,d,pTrip)$
 1: if $dist_E(s,p_{t_1}) > dist_E(d,p_{t_m})$ then
 2:     $d_{max} \leftarrow dist_E(s,p_{t_1})$
 3: else
 4:     $d_{max} \leftarrow dist_E(d,p_{t_m})$
 5: end if
 6: for $i \leftarrow 1$ to $m-1$ do
 7:     $j \leftarrow i+1$
 8:     $d_{ij} \leftarrow ComputeMin(s,d,p_{t_i},p_{t_j})$
 9:     if $d_{ij} + dist_E(p_{t_i},p_{t_j}) > d_{max}$ then
10:         $d_{max} \leftarrow d_{ij} + dist_E(p_{t_i},p_{t_j})$
11:     end if
12: end for
13: repeat
14:     $d_{prev} \leftarrow d_{max}$
15:     $a \leftarrow 2 \times \frac{d_{max}}{(1-e)}$
16:     $O \leftarrow IOR(s,d,a)$
17:     $VG \leftarrow ConstructVG(s,d,p_{t_1},p_{t_2}\ldots,p_{t_m},O)$
18:     $dist_o(s,p_{t_1}) \leftarrow CompObsDist(VG,s,p_{t_1})$
19:     $dist_{sum} \leftarrow 0$
20:     for $i \leftarrow 1$ to $m-1$ do
21:         $j \leftarrow i+1$
22:         $dist_o(p_{t_i},p_{t_j}) \leftarrow CompObsDist(VG,p_{t_i},p_{t_j})$
23:         $dist_{sum} \leftarrow dist_{sum} + dist_o(p_{t_i},p_{t_j})$
24:     end for
25:     $dist_o(d,p_{t_m}) \leftarrow CompObsDist(VG,d,p_{t_m})$
26:     $Tdist_O(s,d,t) \leftarrow dist_o(s,p_{t_1}) + dist_{sum} + dist_o(d,p_{t_m})$
27:     $d_{max} \leftarrow Tdist_O(s,d,pTrip)$
28: until $d_{max} == d_{prev}$
29: return $Tdist_O(s,d,pTrip)$
```

4. AN OTD COMPUTATION TECHNIQUE

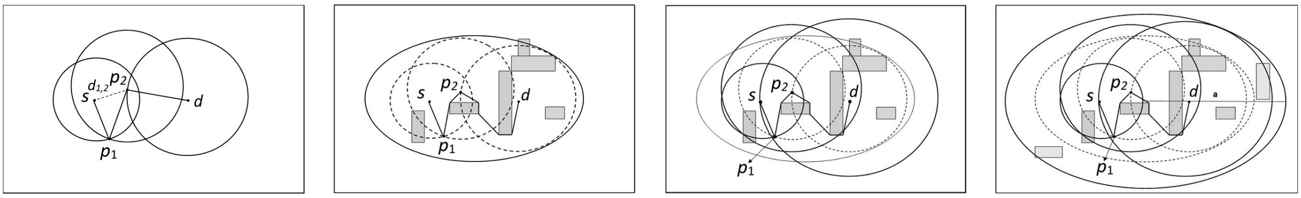
A major challenge of a query processing algorithm in the obstructed space is the complexity of computing the obstructed distance. The obstructed distance is computed as the length of the shortest path between two locations by avoiding the obstacles. There exist algorithms [10] to compute the obstructed distance between two locations. However, computing obstructed distances for pairs of locations independently by applying an existing algorithm requires performing the same computations and the retrieval of same obstacles from the database multiple times. To overcome the limitations, different optimization techniques [2, 9] have been developed in the context of obstructed group nearest neighbor (OGNN) and obstructed reverse nearest neighbor (ORNN) queries, which are not applicable for OOSR queries.

Evaluating an OOSR query requires the computation of a large number of OTDs, and an OTD is the summation of a number of obstructed distances. We develop a novel OTD computation technique that incrementally expands the obstacle retrieval area as an elliptical shape. We develop a technique to compute the length of the major axis of the ellipse to guarantee that obstacles required for every obstructed distance computation are simultaneously retrieved. Furthermore, we reuse the already retrieved obstacles and computed obstructed distances for computing a new OTD. The intuition behind using an elliptical region instead of any other shape is to increase the probability of reusing the already retrieved obstacles, and minimizing the retrieval of obstacles that are not required for obstructed distance computations. We will show in the next section that the refined POI search space in our proposed OOSR algorithms expands as an elliptical region, and therefore there is a high probability that the retrieved POI falls inside the area of the already retrieved obstacles and the obstructed distances involving the POI can be computed using already retrieved obstacles.

We use the existing technique [10] to compute the obstructed distance between two points using a visibility graph. The vertices of a visibility graph are the corner points of polygons representing the obstacles and the locations between which the obstructed distance needs to be computed. There is an edge between two vertices if no obstacle crosses the direct path between those vertices. The obstructed distance between two locations is the length of the shortest path between two vertices representing the locations. It is not feasible to pre-compute a visibility graph for a large set of obstacles. We only retrieve those obstacles from the database that are relevant to the OOSR query and construct the visibility graph.

Algorithm 1 shows the pseudocode for computing an OTD. Without loss of generality, we explain the steps of computing $Tdist_o(s,d,p_1,p_2)$ for an example shown in Figure 2. The algorithm computes $dist_o(s,p_1)$, $dist_o(p_1,p_2)$, and $dist_o(p_2,d)$ simultaneously. Using the function $ComputeMin$ in Line 8, the algorithm finds the Euclidean distance $dist_E(p_2,s)$ as the minimum among $dist_E(p_1,s)$, $dist_E(p_2,s)$, $dist_E(p_1,d)$, and $dist_E(p_2,d)$. Thus, $dist_E(p_2,s)$ is assigned to d_{12} and p_2 becomes the center of the circle used for computing $dist_o(p_1,p_2)$ as shown in Figure 2(a).

In the next step, to compute $dist_o(s,p_1)$, $dist_o(p_1,p_2)$, and $dist_o(p_2,d)$, the algorithm retrieves obstacles inside the circles centered at s, p_2 and d with radius $dist_E(s,p_1)$, $dist_E(p_1,p_2)$, and $dist_E(p_2,d)$, respectively. Figure 2(a) shows that there are overlaps among the circles. Thus, to avoid the retrieval of same POIs multiple times, our algorithm computes an ellipse with foci at s and d that includes three circles, and retrieves obstacles in the ellipse as shown in Figure 2(b). To ensure the inclusion of the circles, the periapsis, i.e., the smallest radial distance of the ellipse needs to be greater than or equal to d_{max}, where d_{max} is the maximum of $dist_o(s,p_1)$, $d_{12} + dist_o(p_1,p_2)$, and $dist_o(p_2,d)$. Thus, the length



(a) (b) (c) (d)

Figure 2: Steps of computing $Tdist_o(s,d,p_1,p_2)$

of the major axis is computed as $2 \times \frac{d_{max}}{(1-e)}$, where eccentricity e is determined in experiments. The function *IOR* in Line 16 incrementally retrieves nearest obstacles with respect to s and d, where the distance is measured as the summation of the minimum Euclidean distances of the obstacle from s and d, respectively.

The algorithm constructs the visibility graph and computes $dist_o(s,p_1)$, $dist_o(p_1,p_2)$, and $dist_o(p_2,d)$ based on the retrieved obstacles. In Figure 2(c), we see that the radius of the circles centered at p_2 and d increases to $dist_o(p_1,p_2)$ and $dist_o(p_2,d)$ from $dist_E(p_1,p_2)$ and $dist_E(p_2,d)$, respectively. Since $dist_o(s,p_1)$ and $dist_E(s,p_1)$ are equal, the circle centered at s does not change and $dist_o(s,p_1)$ is finalized. The algorithm again retrieves obstacles so that the new ellipse includes the circles as shown in Figure 2(d). We observe that in Figure 2(d), though new obstacles are retrieves but those obstacles do not increase any of the obstructed distance. Thus, $dist_o(p_1,p_2)$, and $dist_o(p_2,d)$ are finalized and $Tdist_o(s,d,p_1,p_2)$ is computed in Line 26.

Algorithm 2 RRB_OOSR(s,d,T)

```
Input: A source $s$, a destination $d$, required POI types $T$
Output: The answer set $A$
1: $A_{initial} \leftarrow RetrieveInitialPOIs(s,d,T)$
2: $POITrips \leftarrow CompTrips(A_{initial})$
3: $POITrips_{prev} \leftarrow POITrips$
4: $MinTDist_o \leftarrow \infty$
5: for each $pTrip \in POITrip$ do
6:     $TDist_o \leftarrow CompOTD(s,d,pt)$
7:     if $TDist_o < MinTDist_o$ then
8:         $MinTDist_o \leftarrow TDist_o$
9:         $A \leftarrow pTrip$
10:    end if
11: end for
12: $Maxd \leftarrow FindMaxDist(A_{initial})$
13: if $Maxd < MinTDist_o$ then
14:     $A_{range} \leftarrow RetrievePOIs(s,d,T,MinTDist_o)$
15:     $POITrips \leftarrow CompNewTrips(A_{initial},A_{range},POITrips_{prev})$
16:     for each $pTrip \in POITrips$ do
17:         $TDist_o \leftarrow CompOTD(s,d,pTrip)$
18:         if $TDist_o < MinTDist_o$ then
19:             $MinTDist_o \leftarrow TDist_o$
20:             $A \leftarrow pTrip$
21:         end if
22:     end for
23: end if
24: return $A$
```

5. OOSR ALGORITHMS

In this section, we present efficient algorithms for processing OOSR queries. We develop a pruning technique to refine the POI search space by exploiting the Euclidean lower bound and elliptical properties. A POI outside the refined POI search space cannot provide the minimum OTD. The number of possible trips and OTD computations decrease with the smaller number of retrieved POIs from the database, i.e., the smaller POI search space.

According to the Euclidean lower bound property, the Euclidean trip distance is smaller or equal to the OTD. On the other hand, according to the elliptical property, the Euclidean distance between two foci of an ellipse via a POI outside the ellipse is greater than or equal to the length of the major axis of the ellipse. In our OOSR algorithms, we represent the POI search space using an ellipse, where the foci of the ellipse are at the source and destination locations of a user, and the length of the major axis of the ellipse is equal to the upper bound of the OTD. Thus, POIs outside the ellipse cannot further minimize the OTD.

We propose two OOSR algorithms: *RRB_OOSR* (range retrieval based OOSR) and *IRB_OOSR* (incremental retrieval based OOSR). The key difference between our algorithms, *RRB_OOSR* and *IRB_OOSR*, is that *RRB_OOSR* computes the upper bound of the OTD, refines the POI search space once, and then retrieves all POIs inside the POI search region using a range query. On the other hand, *IRB_OOSR* incrementally retrieves POIs and gradually refines the search space. The advantage of *IRB_OOSR* is that it retrieves less number of POIs than *RRB_OOSR*.

Both *RRB_OOSR* and *IRB_OOSR* use a heuristic [7] to compute the upper bound of the OTD. The heuristic retrieves an initial set of POIs $A_{initial}$ that includes the nearest POI of every required type from s and d. The Euclidean aggregate distance (EAD) of a POI from s and d is computed as the summation of Euclidean distances of the POI from s and d, respectively. In addition to the nearest POI of every required POI type, $A_{initial}$ also includes other POIs of required types that have EAD smaller than or equal to the maximum of EADs of the nearest POIs from every required type.

Algorithm 2 shows the pseudocode for *RRB_OOSR*. The algorithm retrieves initial POIs as $A_{initial}$ using the heuristic (Line 1), computes the sets of possible combinations of POIs as *POITrips* (Line 2), determines the OTD with respect to s and d for every set using Algorithm 1 (Line 6), and finds the upper bound of the OTD as $MinTDist_o$ in Line 8.

Next the algorithm computes *Maxd* in Line 12. The POIs that falls inside the ellipse with foci s and d, and the major axis equal to *Maxd* have been already retrieved. If $Maxd \geq MinTDist_o$, the trip with the minimum OTD has been already found because a POI that has EAD greater or equal to *Maxd* cannot further minimize $MinTDist_o$. Otherwise, the algorithm retrieves all POIs in the refined POI search space (i.e., the POIs whose EADs from s and d are smaller than $MinTDist_o$), computes the set of new combination of POIs by excluding the combinations that have already considered ($POITrips_{prev}$), and finds the set of POIs that provide the minimum OTD with respect to s and d.

Algorithm 3 shows the pseudocode for *IRB_OOSR*. Instead of retrieving all POIs in the refined POI search space, the algorithm incrementally retrieves the next nearest POI with the smallest EAD *Maxd* from s and d (Line 14). After retrieving a new POI, the algorithm further minimizes the upper bound of the OTD as $MinTDist_o$ in Line 20, if possible. The incremental retrieval of POIs continues until the condition $Maxd < MinTDist_o$ is satisfied.

```
Algorithm 3 IRB_OOSR($s, d, T$)
Input: A source $s$, a destination $d$, required POI types $T$
Output: The answer set $A$
$A_{init} \leftarrow RetrieveInitialPOIs(s, d, T)$
$POITrips \leftarrow CompTrips(A_{initial})$
$POITrips_{prev} \leftarrow POITrips$
$MinTDist_o \leftarrow \infty$
for each $pTrip \in POITrips$ do
    $TDist_o \leftarrow CompOTD(s, d, pTrip)$
    if $TDist_o < MinTDist_o$ then
        $MinTDist_o \leftarrow TDist_o$
        $A \leftarrow pTrip$
    end if
end for
$Maxd \leftarrow FindMaxDist(A_{initial})$
while $Maxd < MinTDist_o$ do
    $p \leftarrow RetrieveNextPOI(s, d, T)$
    $Maxd \leftarrow dist(s, p) + dist(p, d)$
    $POITrips \leftarrow CompNewTrips(A_{initial}, p, POITrips_{prev})$
    for each $pTrip \in POITrips$ do
        $TDist_o \leftarrow CompOTD(s, d, pTrip)$
        if $TDist_o < MinTDist_o$ then
            $MinTDist_o \leftarrow TDist_o$
            $A \leftarrow pTrip$
        end if
    end for
end while
return $A$
```

Table 1: Experimental Setup

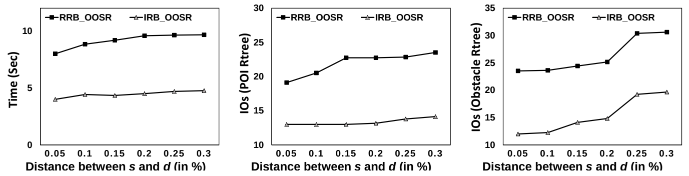
Parameter	Range	Default Value
Distance between s and d	0.05% to 0.3%	0.15%
Total POI types	10, 15, 20, 25, 30	20
Required POI types	1, 2, 3, 4, 5	3

6. EXPERIMENTS

Since we first address OOSR queries, we compare our proposed algorithms through experiments. We vary the distance between s and d, the number of total POI types in the POI data set and the number of required POI types in the query. Table 2 shows the range and default value of each parameter that we used in our experiments. When we vary a parameter in an experiment, we set other parameters to their default values. We used the real dataset of Germany, which consists of 34334 minimum bounded rectangles (MBRs) of railway lines (rrlines) that represent obstacles and 307992 MBRs of hypsography data (hypsogr) that represent POIs in our experiments. We normalized the total space into 10,000 × 10,000 square units. We conducted each experiment for 50 samples of OOSR queries and obtained the average experimental results. We measured the processing time and IO cost using an Intel(R) Core i5-5200U CPU (2.20 GHz) with 4 GB RAM.

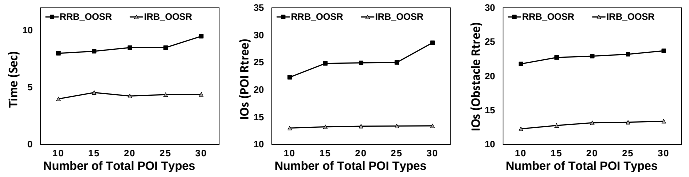
Initially, we varied the values of eccentricity of the ellipse e as 0, 0.25, 0.5, 0.75 and 1, and run experiments for the default values of other parameters. We found that the algorithms perform better in terms of time and IOs for the value of $e = 0.75$. Therefore we set this value as the default eccentricity (e) in our experiments.

Figure 3 shows that the processing time and IOs increases for both of our algorithms with the increase of the distance between s and d. This is because when the distance between s and d increases, the areas for retrieving POIs and obstacles also increase. We also observed that *IRB_OOSR* performs better in terms of both time and IO cost than *RRB_OOSR*. This is expected as *RRB_OOSR* retrieves more POIs than *IRB_OOSR*. Figures 4 and 5 show the similar trends for varying the number of total POI types and the number of required POI types, respectively.



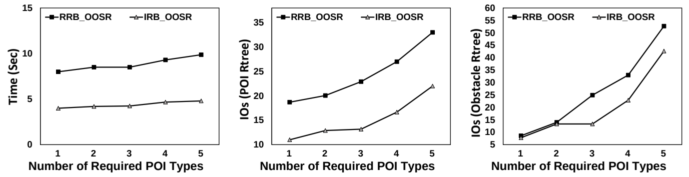
(a) (b) (c)

Figure 3: Effect of the distance between s and d in %



(a) (b) (c)

Figure 4: Effect of the number of total POI types



(a) (b) (c)

Figure 5: Effect of the number of required POI types

7. CONCLUSION

We developed a novel OTD computation technique, and OOSR algorithms: *RRB_OOSR* and *IRB_OOSR*. Experiments show that our approach can evaluate OOSR queries in real time, and on average *IRB_OOSR* requirers 2.1 times less processing time and 1.7 times less IOs than *RRB_OOSR* to process OOSR queries.

Acknowledgments

This research is supported by Bangladesh University of Engineering and Technology (BUET) and University of Asia Pacific (UAP).

8. REFERENCES

[1] H. Chen, W. Ku, M. Sun, and R. Zimmermann. The multi-rule partial sequenced route query. In *SIGSpatial*, pages 10:1–10:10, 2008.

[2] Y. Gao, J. Yang, G. Chen, B. Zheng, and C. Chen. On efficient obstructed reverse nearest neighbor query processing. In *GIS*, pages 191–200, 2011.

[3] Y. Gao, B. Zheng, G. Chen, C. Chen, and Q. Li. Continuous nearest-neighbor search in the presence of obstacles. *ACM Trans. Database Syst.*, 36(2):9:1–9:43, 2011.

[4] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S. Teng. On trip planning queries in spatial databases. In *SSTD*, pages 273–290, 2005.

[5] Y. Ohsawa, H. Htoo, N. Sonehara, and M. Sakauchi. Sequenced route query in road network distance based on incremental euclidean restriction. In *DEXA*, pages 484–491, 2012.

[6] M. Sharifzadeh, M. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. *The VLDB Journal*, 17(4):765–787, 2008.

[7] S. C. Soma, T. Hashem, M. A. Cheema, and S. Samrose. Trip planning queries with location privacy in spatial databases. *World Wide Web*, 2016.

[8] N. Sultana, T. Hashem, and L. Kulik. Group nearest neighbor queries in the presence of obstacles. In *SIGSPATIAL*, pages 481–484, 2014.

[9] N. Sultana, T. Hashem, and L. Kulik. Group meetup in the presence of obstacles. *Inf. Syst.*, 61:24–39, 2016.

[10] J. Zhang, D. Papadias, K. Mouratidis, and M. Zhu. Spatial queries in the presence of obstacles. In *EDBT*, pages 366–384, 2004.

Demonstration

I^2: Interactive Real-Time Visualization for Streaming Data

Jonas Traub
Technische Universität Berlin
jonas.traub@tu-berlin.de

Nikolaas Steenbergen
German Research Center for Artificial Intelligence (DFKI)
nikolaas.steenbergen@dfki.de

Philipp M. Grulich
German Research Center for Artificial Intelligence (DFKI)
philipp.grulich@dfki.de

Tilmann Rabl
Technische Universität Berlin
rabl@tu-berlin.de

Volker Markl
Technische Universität Berlin
volker.markl@tu-berlin.de

ABSTRACT

Developing scalable real-time data analysis programs is a challenging task. Developers need insights from the data to define meaningful analysis flows, which often makes the development a trial and error process. Data visualization techniques can provide insights to aid the development, but the sheer amount of available data frequently makes it impossible to visualize all data points at the same time. We present I^2, an interactive development environment that coordinates running cluster applications and corresponding visualizations such that only the currently depicted data points are processed and transferred. To this end, we present an algorithm for the real-time visualization of time series, which is proven to be correct and minimal in terms of transferred data. Moreover, we show how cluster programs can adapt to changed visualization properties at runtime to allow interactive data exploration on data streams.

1. INTRODUCTION

The amount of available real-time data increases rapidly with the growth of the Internet of Things. Such data is provided in the form of continuous data streams and includes various kinds of information such as stock prices, Twitter messages, Wikipedia edits, weather data, and GPS positions. Systems such as Apache Spark and Storm can process huge amounts of data with low latencies in a cluster to provide real-time analysis. Nevertheless, the development of analysis programs for these platforms remains a complex task, which requires insights about the processed data.

A visualization of the incoming datastream can provide such insights, but visualizing big data in real-time is a challenge itself. Since display capabilities are limited to a certain plot resolution (height and width of the screen) and local processing capabilities (e.g., a browser), it is usually impossible to show all individual data points from a high bandwidth data stream. For example, even though a time series may consist of 2000 measurements per second, the visualization of a second in a line chart is limited to a certain amount of pixel columns. Thus, a user has to trade off between the length of the shown history (time span covered on the time axis) and the resolution of the provided plot (available pixel columns per time) as shown in Figure 1a.

Interestingly, it is proven that the amount of data which is required to plot a correct line chart depends only on the number of pixel columns and not on the data. Jugel et al. [8] derive standard SQL queries from a given plot resolution and provide a loss-free plot from only four values per pixel column which reduces the computational load of the system. We show how the same values can be computed in a parallel dataflow program to allow the live visualization of incoming streaming data. Additionally, we take care of differences between event time and processing time as well as tuples arriving out-of-order, which makes processing streaming data a more complex task.

We integrate the efficient live visualization of time series as line chart together with other types of visualizations in I^2, our interactive development environment which connects distributed data analysis programs with the visualization of the results.[1] The name I^2 emphasizes two types of interactivity: *(i)* through code changes and *(ii)* through an interactive visualization GUI. With I^2, developers can change and deploy the code of analysis pipelines and corresponding result visualizations in a one-click fashion. Moreover, running applications adapt to changes in the visualization, e.g., if the user zooms into a map, and ensure that only the data points which are depicted in the current visualization are processed and transferred towards the front end. As a result, I^2 decreases the workload in the cluster backend as well as the visualization front end. Summarizing, our contributions are:

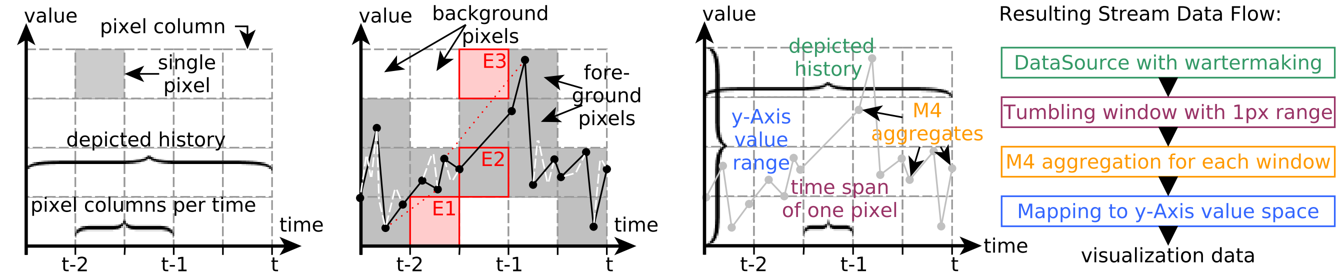
1. We present an interactive environment for visualization supported development of streaming cluster applications.
2. We show that our solution significantly reduces the amount of processed and transferred data while still providing loss-free visualizations.
3. We provide an algorithm for the live visualization of time series in line charts, which is proven to be correct and minimal in terms transferred data.

In our demonstration, we use I^2 for real-time event-based sport analytics. We therefore explore a data set from the DEBS 2013 Grand Challenge [10] consisting of more than 2.6 GB sensor data recorded at a football match with up to 2000Hz sampling rates. The data provides detailed real-time information about all players as well as the ball.

[1] https://github.com/TU-Berlin-DIMA/i2

Series ISSN: 2367-2005

10.5441/002/edbt.2017.61



(a) The tradeoff between depicted history and plot precision.

(b) The M4 aggregation technique for time-series data.

(c) Deriving a stream data flow program for the real-time visualization of time-series data with M4.

Figure 1: Efficient real-time visualization of time series data.

Related Work. In contrast to existing data exploration techniques [7], our demonstration combines three functionalities within a single environment: *(i)* the rapid development and deployment of cluster applications with streaming data, *(ii)* the automatic adaptation of running cluster jobs to changed visualization properties, and *(iii)* the efficient reduction of data to prevent overload of the visualization front-end. While other solution require an additional intermediate layer between database and visualization [4], I^2 directly integrates into data analysis applications. Other approaches like [1, 2, 9, 12] use sampling strategies for fast visualizations of huge amounts of data, but as opposed to I^2 disregard physical display properties and do not cover live plots of streaming data. Wu et al. [14] take into account visualization properties and automatically derive SQL-queries, but use a domain specific language. In contrast, I^2 works with any query language integrated in Apache Zeppelin.

In the remainder of this paper, we first present our solution for the visualization of time-series in line charts in Section 2. We then present the over-all architecture of I^2 in Section 3 and our demonstration in Section 4.

2. VISUALIZATION OF TIME SERIES

High volume time series data is omnipresent in many domains such as banking, weather data, facility monitoring, or, as in our demonstration, sport analytics. A naive approach for the visualization of time series would send all available data points towards the front end, which causes the visualization to crash in case the amount of input data increases as we will show in Section 4. The M4 aggregation technique [8] overcomes this limitation and constantly transfers just four values per pixel column. Furthermore, M4 is proven to provide loss-free plots compared to plots of the original data.

Figure 1b illustrates the functioning of the M4 aggregation. For each pixel column, M4 finds the minimum and maximum value as well as the first and the last value (minimum and maximum timestamp). All pixels which are crossed by the line connecting the extracted data points are colored and thus become foreground pixels. The intuitive approach to take only the minimum and maximum values into consideration would be insufficient. This would result in the red dotted line in Figure 1b and cause the pixel errors E1, E3 (wrongly colored) as well as E2 (not colored).

In I^2, we want to visualize streaming data in real-time. While M4 only considers finite data stored in a relational database, the real-time requirement adds several new challenges: instead of standard SQL queries, we now need *parallelizable processing pipelines.* Due to network delays and failures, there might be a gap between event time (the point in time a measure is taken) and processing time (the point in time the data is processed). Since data points may arrive out-of-order, we can never guarantee that the data for a pixel column is complete and possibly need to update past pixel columns in case of delayed input data. We address these challenges, as we derive a complete stream processing pipeline from a given plot resolution and the length of the depicted history as shown in Figure 1c. The pipeline mainly consists of four steps each of which can be executed as an operator with possibly multiple parallel instances.

Watermarks. Watermarks flow through the pipeline alongside the regular data and propagate the progress of event time. A watermark of time t_w means that no later processed event will have a timestamp $t_e < t_w$. We input watermarks at the data source of our pipeline to mark the smallest timestamp which is still covered by the live plot. Hence, we update pixel columns in case data arrives out-of-order. However, we avoid unnecessary processing of out-of-order data which arrives so late that the corresponding pixel column of the live chart is no longer displayed.

Windowing. We apply a time window function which splits the stream into finite data chunks spanning the time of one pixel column. We then compute the M4 aggregates over these windows and respectively for each pixel column. For the lack of space, we omit further details about the processing of out-of-order events and refer the reader to [5].

Value compression. Finally, we map the results of the aggregation to the value space of the y-axis which allows us to represent each value with less bytes.

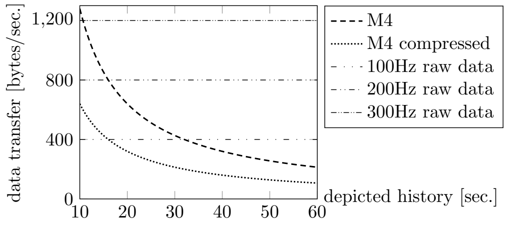


Figure 2: The required bandwidth for an 800x600px plot.

Figure 2 shows the savings in the input bandwidth of the visualization assuming an 800x600px plot showing 4 byte integer values. Note that the bandwidth required by M4 is independent from the frequency of the underlying raw data and solely depends on the length of the depicted history. The longer the depicted history, the more data is aggregated into one pixel column, which causes the required bandwidth to

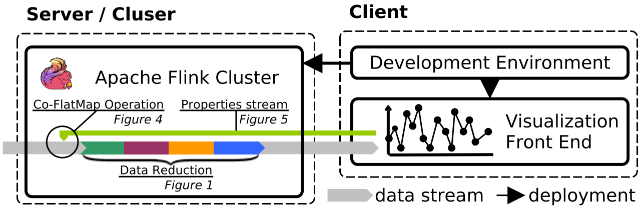


Figure 3: I^2 architecture overview.

decrease. In the next section, we show how our streaming ready M4 aggregation pipeline is integrated into the overall architecture of the I^2 development environment.

3. I² DEVELOPMENT ENVIRONMENT

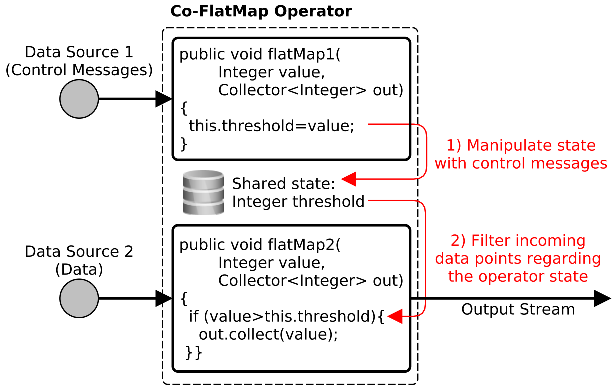
The I^2 development environment aims to seamlessly connect live data visualization with the development of streaming data analysis pipelines. We, therefore, directly link a development environment and result visualizations within in a single front end (Figure 3). Developers can deploy data analysis pipelines as well as visualizations in a one-click fashion. While the visualization is provided within the same GUI as the code editor, the analytics pipeline is deployed on an Apache Flink cluster to be capable of processing high bandwidth streams in parallel.

Apache Flink [3, 5] is an open source platform for big data batch and stream processing. The basis of Flink is a fault tolerant execution engine. Programs are represented as operator graphs and the full processing pipeline is executed concurrently. Thus, the output tuples of an operator can be processed immediately by succeeding operators. Flink allows operators to have state. An asynchronous snapshot algorithm [6] ensures exactly once processing guarantees even in case of failures. Flink fits perfectly to I^2 since we need stateful operators to store current visualization parameters and low latency processing to quickly adapt running jobs to changes.

Apache Zeppelin. The I^2 front end is based on Apache Zeppelin, but was extended to support automatic data reduction depending on current visualization parameters. In general, Zeppelin aims to support quick development of programs, enabling interactive analytics in web based notebooks. It is similar to IPython [11], but focuses on large scale datasets and distributed computing. Zeppelin notebooks are data driven, interactive, and can be edited collaboratively by multiple users. Moreover, Zeppelin supports a variety of execution back ends. Zeppelin is not limited to classical dashboards; it also allows to develop source code, submit jobs directly to the cluster, and retrieve results immediately.

Runtime Adaptive Operators. I^2 informs running Flink jobs about changes of the visualization parameters. For example, if the user zooms into a map or changes the length of the depicted history of a time series plot. The running cluster program has to adapt to such changes with low latency in order to immediately provide the required data for the visualization. Since a redeployment of a job in the cluster can take more than a minute, we need to adapt jobs at runtime.

We push changes of the visualization parameters as control messages in a separate stream to the running Flink job. Only the type of an operator (e.g., filter or aggregation) is



```
public void flatMap1(
        Integer value,
        Collector<Integer> out)
{
  this.threshold=value;
}
```

```
public void flatMap2(
        Integer value,
        Collector<Integer> out)
{
  if (value>this.threshold){
    out.collect(value);
 }}
```

Figure 4: A runtime adaptive filter operator for variable thresholds in Apache Flink.

defined a priori, while we allow to adjust the parameters of the operator (e.g., filter predicate or aggregate function) on the fly at runtime. We use Flink's CoMap operators to process the control messages and the actual data points together in a shared runtime adaptive operator.

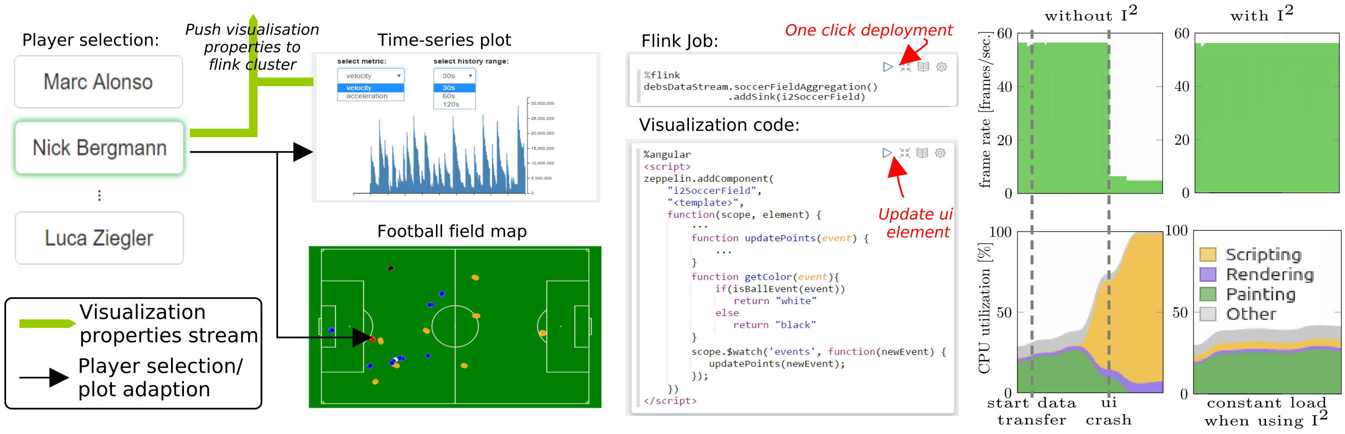
Flink's CoMap operators consume two input streams while input items from each stream are processed by separate *user defined functions* (UDFs). Nevertheless, both UDFs can access a shared operator state which is used to communicate between them. Figure 4 shows how we can utilize a CoFlatMap operator to adapt to changed properties: in this example, one input stream consists of control messages containing changes to the threshold of a filter operation. The responsible UDF saves the current threshold as operator state (Figure 4, 1). Each value from the actual data stream is compared to the currently stored threshold and all smaller values are filtered out (Figure 4, 2). In general, arbitrary changes to a selection criteria, aggregation function, windowing semantics, and other operations are possible using this architecture.

4. DEMONSTRATION

In our demonstration, we allow the visitor to experience the fast visualization supported development with I^2. This covers the development of the Flink job running in the cluster as well as changing the visualizations. At the same time, we continuously show the savings in terms of the transferred data volume which are archived by I^2. When we increase the data rates of the input streams, I^2 will hide that workload from the visualization while without using I^2 the front end would first become unresponsive and finally crash.

Data. We replay the data set which was provided with the DEBS Grand Challenge 2013 [10]. This data set consists of sensor data, which was recorded at a football match. The speed, acceleration, and position of the ball are tracked with a frequency of 2000Hz. In addition, each player has two sensors close to his shoes which are tracked with a 200Hz frequency. In total, roughly 15.000 data points are provided for each second of the match.

Demonstration. We show an interactive dashboard to analyze the performance of individual players in detail. Users can either select a player manually or automatically follow the ball possession, which involves detecting peaks in the measures of the ball sensor as well as correlating these peaks with the data from the player sensors. Our dashboard shows



Flink Job:

```
%flink
debsDataStream.soccerFieldAggregation()
               .addSink(i2SoccerField)
```

Visualization code:

```
%angular
<script>
zeppelin.addComponent(
    "i2SoccerField",
    "<template>",
    function(scope, element) {
        ...
        function updatePoints(event) {
            ...
        }
        function getColor(event){
            if(isBallEvent(event))
               return "white"
            else
               return "black"
        }
        scope.$watch('events', function(newEvent) {
           updatePoints(newEvent);
        });
    })
</script>
```

(a) Interactive Dashboard

(b) Development Environment

(c) Performance Monitoring

Figure 5: Selected screenshots from the I^2 demonstration.

different metrics (e.g, acceleration and speed) for the selected player as well as the player's current position on the football field (Figure 5a).

We first try to run our dashboard without using I^2, meaning that no data reduction is applied and all data - roughly 15.000 tuples/sec. - is transferred towards the frontend. As shown in Figure 5c (left), the UI works only for a short moment before it becomes unresponsive due to a CPU overload.

We now run the same dashboard with I^2, pushing the current visualization properties to the running Flink job as described in Section 3. This information is then used by Flink to apply different data reduction techniques: knowing the currently selected player enables adaptive filtering as shown in Figure 4 and knowing the plot resolution of line charts allows to apply the M4 aggregation technique we presented in Section 2. The soccer field map combines different data reduction techniques. We reduce the precision of the position reports based on the plot resolution and at the same time apply load shedding [13] to reduce the data rate to the current frame rate of the visualization. As shown on Figure 5c (right), the presented dashboard runs fluently when using I^2 with close to 60 frames per second and a CPU utilization below 50%.

Interactivity. We demonstrate the two types of interactivity provided by I^2. First, we show that visualization properties can be changed easily in the dashboard and that the running Flink job adapts with low latency to e.g., changes in the player selection or the length of the depicted history of line charts.

Second, we demonstrate the interactivity through code changes. Interactive code changes allow an even more flexible data exploration and the rapid development of cluster applications. We first show how the code for the visualizations can be adapted and directly deployed without a need to restart the running Flink job. We then show how we can connect an additional data source for twitter messages and how these messages can be correlated to the data we used before. The extended Flink job is directly deployed to the cluster with just one click.

Evaluation. We exemplary compared the performance of I^2 for the dashboard described above (Figure 5c). Our experiment showed that the amount of transferred data, the memory utilization, the CPU load, and the frame rate remain constant throughout the game when I^2 is active. Switching of I^2 causes the visualization to become unresponsive immediately due to the massive amount of arriving data. With activated I^2, the bottleneck is no longer the visualization, but the power of the used Flink cluster.

5. CONCLUSIONS

I^2 enables two types of interactivity: first, the user can specify real-time analysis programs and change them on the fly. Second, the interactive visualization of the results adapts currently running cluster applications without a need to restart. Using I^2, the amount of data points to be processed and transferred to the front end can be reduced significantly without quality loss, enabling the live visualization of high bandwidth data streams. The capabilities of I^2 have been demonstrated in an interactive example using real-world data.

Acknowledgements. This work was supported by the EU Horizon 2020 project Streamline (688191) and the German Ministry for Education and Research as Berlin Big Data Center (01IS14013A) and Software Campus (01IS12056).

6. REFERENCES

[1] S. Agarwal et al. BlinkDB: queries with bounded errors and bounded response times on very large data. EuroSys, 2013.

[2] S. Agarwal et al. Knowing when you're wrong: building fast and reliable approximate query processing systems. SIGMOD, 2014.

[3] A. Alexandrov et al. The stratosphere platform for big data analytics. *VLDBJ*, 2014.

[4] L. Battle et al. Dynamic reduction of query result sets for interactive visualizaton. IEEE BigData, 2013.

[5] P. Carbone et al. Apache Flink^{TM}: Stream and batch processing in a single engine. *IEEE Data Eng. Bulletin*, 2015.

[6] P. Carbone et al. Lightweight asynchronous snapshots for distributed dataflows. *arXiv*, 2015.

[7] S. Idreos et al. Overview of data exploration techniques. SIGMOD, 2015.

[8] U. Jugel et al. M4: a visualization-oriented time series data aggregation. *VLDB*, 2014.

[9] A. Kim et al. Rapid sampling for visualizations with ordering guarantees. 2015.

[10] C. Mutschler et al. The DEBS 2013 grand challenge. 2013.

[11] F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *CISE*, 2007.

[12] L. Sidirourgos et al. Scientific discovery through weighted sampling. IEEE BigData, 2013.

[13] N. Tatbul et al. Load shedding in a data stream manager. VLDB, 2003.

[14] E. Wu et al. The case for data visualization management systems: vision paper. *VLDB*, 2014.

HDM: Optimized Big Data Processing with Data Provenance

Dongyao Wu [†,‡], Sherif Sakr [†,‡,*], Liming Zhu [†,‡]
[†]Data61, CSIRO, Sydney, Australia
[‡]School of Computer Science and Engineering, University of New South Wales, , Sydney, Australia
[*]King Saud bin Abdulaziz University for Health Sciences, National Guard, Riyadh, Saudi Arabia
{firstname.lastname}@data61.csiro.au

ABSTRACT

Big Data applications are becoming more complex and experiencing frequent changes and updates. In practice, manual optimization of complex big data jobs is time-consuming and error-prone. Maintenance and management of evolving big data applications is a challenging task as well. We demonstrate HDM, Hierarchically Distributed Data Matrix, as a big data processing framework with built-in data flow optimizations and integrated maintenance of data provenance information that supports the management of continuously evolving big data applications. In HDM, the data flow of jobs are automatically optimized based on the functional DAG representation to improve the performance during execution. Additionally, comprehensive meta-data related to explanation, execution and dependency updates of HDM applications are stored and maintained in order to facilitate the debugging, monitoring, tracing and reproducing of HDM jobs and programs.

Keywords

Big Data; Data Flow Optimization; Provenance Management

1. INTRODUCTION

We are experiencing the era of big data that has been fuelled by the striking speed of the growth in the amount of data that has been generated and consumed. Several big data processing frameworks (e.g., MapReduce [2], Spark [6] and Flink [1], etc.) have been introduced to deal with the challenges of processing the ever larger data sets [3]. These frameworks significantly reduce the complexity of writing large scale data-oriented applications. However, in practice, as big data programs and applications have become more and more complicated, it is almost impossible to manually optimize the performance of programs written by diversified programmers. Therefore, built-in optimizers are crucial for tackling the challenges of improving the performance of executing those hand-written programs and applications. At the same time, realistic data analytics applications are continuously evolving in order to deal with the non-stop changes in the real world. In practice, managing and analyzing those continuously evolving big data applications have resulted in big technical debts [4]. Therefore, there are increasing requirements for data provenance to support analyzing, tracing and reproduction of historical versions of data analytics applications.

In this paper, we demonstrate HDM, (Hierarchically Distributed Matrix) [5], a big data processing framework with built-in data optimizations for execution and data provenance supports for managing continuously evolving big data applications. In particular, HDM is a lightweight, functional and strongly-typed data representation which contains complete information (such as data format, locations, dependencies and functions between input and output) to support parallel execution of data-driven applications [5]. Exploiting the functional nature of HDM enables deployed applications of HDM to be natively integrable and reusable by other programs and applications. In addition, by analyzing the execution graph and functional semantics of HDMs, multiple optimizations are provided to automatically improve the execution performance of HDM data flows. Moreover, by drawing on the comprehensive information maintained by HDM graphs, the runtime execution engine of HDM is also able to provide provenance and history management for submitted applications.

2. HDM FRAMEWORK

2.1 System Overview

Fig 1 shows the system architecture of the HDM runtime engine which is composed of three main components:

- *Runtime Engine*: is responsible for the management of HDM jobs such as explaining, optimization, scheduling and execution. Within the runtime engine, the AppManager manages the information of all deployed jobs. TaskManager maintains the activated tasks for runtime scheduling in the Schedulers; Planner and Optimizers interpret and optimize the execution plan of HDMs in the explanation phases; HDM manager manages the information and states of the HDM blocks in the entire cluster; Execution Context is an abstraction component to support the execution of scheduled tasks on either local or remote nodes.
- *Coordination Service*: is composed of three types of coordinations: cluster coordination, block coordination

Series ISSN: 2367-2005

10.5441/002/edbt.2017.62

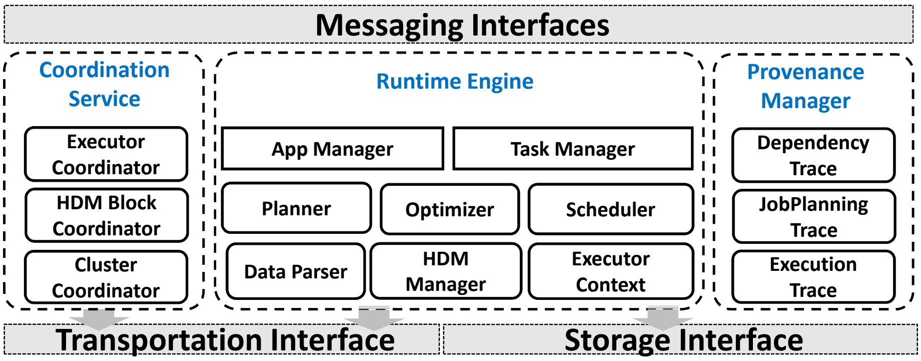


Figure 1: System Architecture of HDM Framework.

and executor coordination. They are responsible for the coordination and management of node resources, distributed HDM data blocks and executors on workers, respectively.

- *Data Provenance Manager*: is responsible to interact with the HDM runtime engine to collect and maintain data provenance information (such as DependencyTrace, JobPlanningTrace and ExecutionTrace) for HDM applications. Those information can be queried and obtained by client programs through messages for the usage of analysis or tracing.

2.2 HDM Data Flow Optimization

One key feature of HDM is that, the execution engine contains built-in planners and optimizers to automatically optimize the functional data flow of submitted applications and jobs. During explanation of HDM applications, the data flow are represented as DAGs with functional dependencies among operations. The HDM optimizers traverse through the DAG to reconstruct and modify the operations based on optimization rules to obtain more optimal execution plans. Currently, the optimization rules implemented in the HDM optimizers include: function fusion, local aggregation, operation reordering and data caching for iterative jobs [5].

- *Function fusion.* During optimization, the HDM planner combines the lined-up non-shuffle operations into one operation with high-order function so that the sequence of operations can be compute within one task rather than separate ones to reduce redundant intermediate results and task scheduling. This rule can be applied recursively on a sequence of fusible operations to form a compact combined operation.

- *Local Aggregation.* Shuffle operations are very expensive in the execution of data-intensive applications. If a shuffle operation is followed with some aggregations, in some cases, the aggregation or part of the aggregation can be applied before the shuffling stage. During optimization, HDM planer tries to move those aggregation operations forward before the shuffling stage to reduce the amount of data that needs to be transferred during shuffling.

- *Operation reordering/reconstruction.* Apart from aggregations, there are a group of operations which filter out a subset of the input during execution. Those operations are called pruning operations. The HDM planner attempts to lift the priority of the pruning operations while sinking the priority of shuffle-intensive operations to reduce the data size that needs to be computed and transferred across the network.

- *Data Caching.* For many complicated and pipelined analytics jobs (such as machine learning algorithms), some intermediate results of the job could be reused multiple times by the subsequent operations. Therefore, it is necessary to cache those repetitively used data to avoid redundant computation and communication. In this case, HDM planner counts the reference for the output of each operation in the functional DAG to detect the potential points that intermediate results should be cached for reusing by subsequent operations.

During optimization process, the rule above are applied one by one to reconstruct the HDM DAG and the optimization can last multiple iterations until there is no change in the DAG or it has reached the maximum number of iterations. The HDM optimizer is also designed to be extendable by adding new optimization rules by developers when it is needed.

2.3 Data Provenance Supports in HDM

It is normally tedious and complicated to maintain and manage applications that are continuously evolving and being updated. In HDM, drawing on comprehensive metadata information maintained by HDM models, the runtime engine is able to provide data provenance supports including execution tracing, version control and job replay in the dependency and execution history management component.

Basically, the HDM server maintains three types of metadata about each submitted HDM jobs including ExecutionTrace, JobPlanningTrace and DependencyTrace.

- *DependencyTrace.* For every submitted HDM program, the server stores and maintains the dependent libraries required for execution. The dependencies and update history are maintained as a tree structure. Based on this information, users are able to reproduce any version of the submitted applications in the history.

- *JobPlanningTrace.* The HDM server also stores the explanation and planning traces for every HDM applications. JobPlanningTrace includes the logical plan, optimizations applied and final physical execution plan after being parallelized.

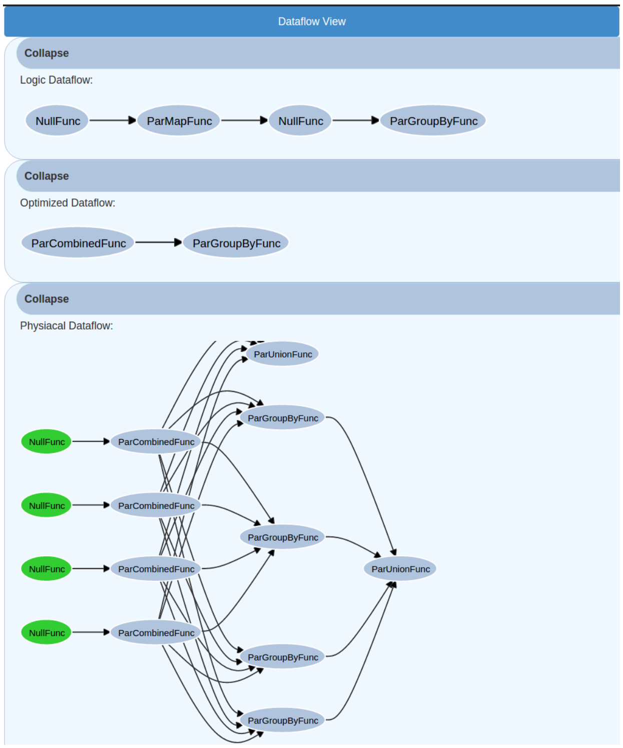


Figure 2: Dataflow Visualization of HDM Applications.

- *ExecutionTrace.* During execution, the HDM server also maintains all the runtime information (execution location, input/output, timestamps and execution status, etc.) related to each executed task and job. These information are very meaningful to monitor and trace back the process of execution of historical jobs and applications.

Drawing on the three types of information maintained in the HDM server, client-side programs can send messages to query and obtain the history and provenance information, so that users and administrators can profile, debug and apply analysis to the deployed applications throughout their life cycles.

3. DEMONSTRATION SCENARIOS

In this demonstration, we will present to the audience the HDM framework[1] from four main aspects: cluster resource monitoring, visualisation data flow optimization, execution history tracing, version-control and dependency management. The demonstration will be conducted on AWS EC2 with one M3.Large instance as the master and 10 nodes M3.XLarge instances as the workers.

To show how HDM optimizes the data flow and provides data provenance support for its applications, we will present an example of Twitter analysis scenario that consists of the following two Tweets analysis programs[2]:

- The first program, presented in Listing 1, looks for the Tweets that are related to recent election events by checking the hashtag of the input Tweets.
- The second program, presented in Listing 2, finds out the Tweets that are related to two candidates: “Trump” and “Hillary” and count the amount for each of them.

Listing 1: Code Snippet of Finding out Tweets

```
val input = HDM("hdfs://10.10.0.100:9091/user/tweets")
val tweets = input.map{ line =>
          val seq = line.split(",")
          Tweet(seq)
      }
val grouped = tweets.groupBy(t => t.hashTag)
val results = grouped.findByKey(_.contains("election"))
```

Listing 2: Code Snippet of Hashtag Counting for Interested Tweets

```
val input = HDM("hdfs://10.10.0.100:9091/user/tweets")
val tweets = input.map{ line =>
          val seq = line.split(",")
          Tweet(seq)
      }
val grouped = tweets.groupBy(t => t.hashTag)
val trumpN = grouped.findByKey(_ == "Trump").count
val hillaryN = grouped.findByKey(_ == "Hillary").count
println(trumpN / hillaryN)
```

Cluster Resource Management. In the first part of the demo, we will show the cluster resource monitor of the HDM manager. The HDM server maintains the resource-related information of all the workers within the cluster. In the HDMConsole, it is able to monitor the resource utilization information (such as CPU, Memory, Network and JVM) for each worker in real time. Therefore, cluster administrator is able to use these information and easily supervise and understand the status of every worker as well as the entire cluster.

Dataflow Optimizations. The second part of the demo shows how the Tweets programs are represented in the HDM DAG and how it is explained, optimized and parallelized by the planner.

- For the first program, the HDM optimizer applies operations reordering to lift the pruning operation `findByKey` to be in front of the shuffle operation `groupBy`. Then the optimizer applies function fusion rule to combine `map` and `findBy` into a single composite operation.
- For the second program, the HDM optimizer applies operation reordering to move the `findByKey` operation to be in front of `groupBy` then applies local aggregation `count` by adding local count in front of `groupBy`. Lastly, it detects the input tweets that are reused by two operations so that the optimizer can add a cache point after the compute operation that generates the output of `tweets`.

The HDM server maintains all the related meta-data (such as the creator, original program, logical plan, physical plan, etc.) to all the submitted HDM applications. In the demonstration, the HDM console visualizes the original logical flow, optimized logical flow and parallelized physical graph

[1] The source code of the HDM framework is available on https://github.com/dwu-csiro/HDM

[2] A demonstration screencast is available on https://youtu.be/Gsz7z5bQ1zI

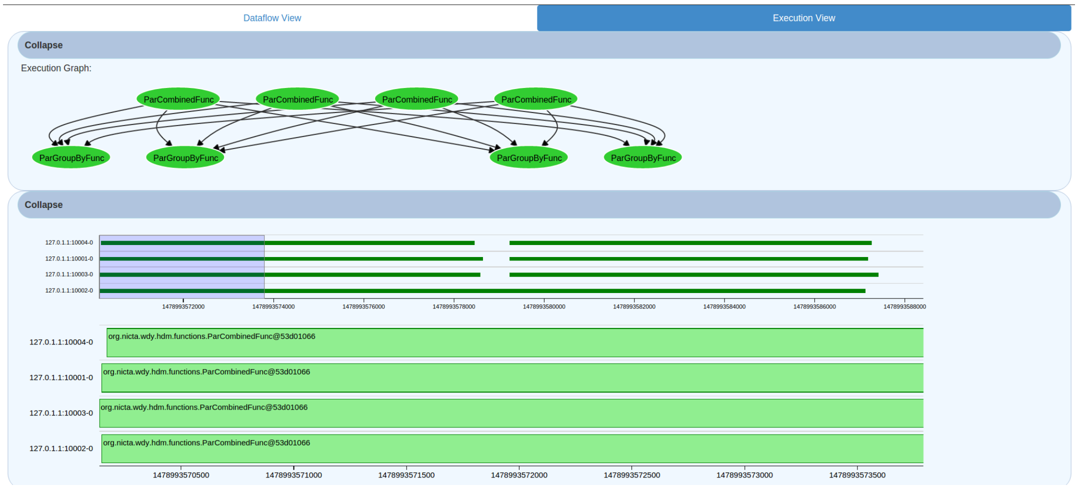


Figure 3: Execution Traces of HDM Applications.

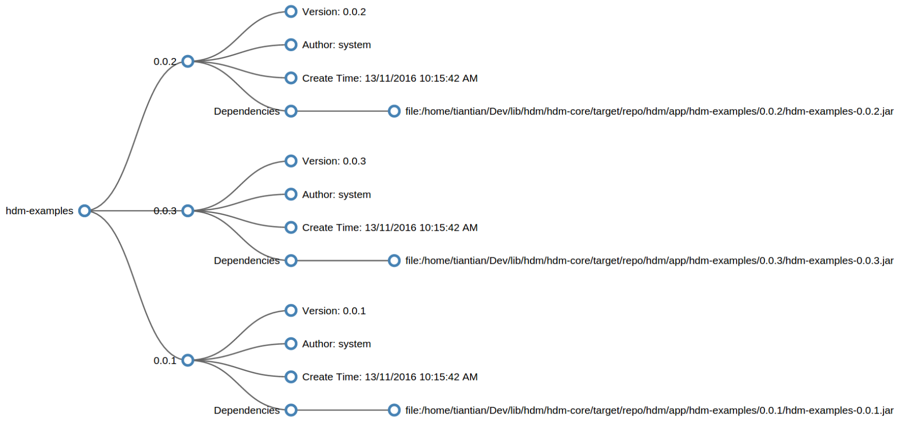

hdm-examples
0.0.2
Version: 0.0.2
Author: system
Create Time: 13/11/2016 10:15:42 AM
Dependencies
file:/home/tiantian/Dev/lib/hdm/hdm-core/target/repo/hdm/app/hdm-examples/0.0.2/hdm-examples-0.0.2.jar
0.0.3
Version: 0.0.3
Author: system
Create Time: 13/11/2016 10:15:42 AM
Dependencies
file:/home/tiantian/Dev/lib/hdm/hdm-core/target/repo/hdm/app/hdm-examples/0.0.3/hdm-examples-0.0.3.jar
0.0.1
Version: 0.0.1
Author: system
Create Time: 13/11/2016 10:15:42 AM
Dependencies
file:/home/tiantian/Dev/lib/hdm/hdm-core/target/repo/hdm/app/hdm-examples/0.0.1/hdm-examples-0.0.1.jar

Figure 4: Dependency Management and Version Control of HDM.

for each execution instances of the HDM applications (Figure 2).

Execution History Tracing. In the third part of the demo, we will show how the execution process can be tracked during and after execution. The HDM server collects and stores the runtime information for each execution task and structures them into DAG based on the task dependencies. During or after the execution of the tasks, the HDM server also updates the status in the stored meta-data when it has received the notification messages. The HDM console also summarizes those information and presented it into a view of execution lanes for each core of the workers (Figure 3).

Dependency Management and Version Control. In the last part of the demo, we will show how the HDM server manages the dependencies and provides version control for submitted applications. The dependency and history manager stores all the updating history of each HDM applications and organizes them into a tree based structure. As a result, administrator users are able to query, analyze and reproduce the historical HDM applications using those dependencies information (Figure 4).

Besides the framework demonstration, we will also discuss in more details about the design choices that we have made on defining the different components of the framework. In addition, performance comparison with the Spark framework [6], using the example scenario, will be presented to demonstrate the efficiency of the HDM optimization techniques.

4. REFERENCES

[1] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink^{TM}: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[2] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[3] S. Sakr. *Big Data 2.0 Processing Systems - A Survey.* Springer Briefs in Computer Science. Springer, 2016.

[4] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning*, 2014.

[5] D. Wu, S. Sakr, L. Zhu, and Q. Lu. Composable and Efficient Functional Big Data Processing Framework. In *IEEE Big Data*, 2015.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

ChaseFUN: a Data Exchange Engine for Functional Dependencies at Scale

Angela Bonifati
University of Lyon 1
angela.bonifati@univ-lyon1.fr

Ioana Ileana
Paris Descartes University
ioana.ileana@parisdescartes.fr

Michele Linardi
Paris Descartes University
michele.linardi@parisdescartes.fr

ABSTRACT

Despite their wide use and importance, target functional dependencies (fds) are still a bottleneck for the state-of-the-art Data Exchange (DE) engines. The consequences range from incomplete support to support at the expense of an important overhead in performance. We demonstrate here ChaseFUN, a DE engine that succeeds in effectively mitigating and taming this overhead, thus making target fds affordable even for very large-sized, complex scenarios. ChaseFUN is a custom chase-based system that essentially relies on exploiting chase step ordering and constraint interaction, so as to piecemeal process, parallelize and dramatically speed-up the chase. Interestingly, the structures and concepts at the core of our system moreover allow it to seamlessly uncover a range of usually opaque details of the chase. As a result, ChaseFUN's two main strengths are: (i) its significant scalability and performance and (ii) its ability to provide detailed, granular insight on the DE process. Across our demonstration scenarios, we will emphasize our system's practical performance and ability to scale to very large source instances and sets of constraints. Furthermore, we will aim at providing the user with a novel, behind-the-scenes view on the internals of the ongoing chase process, as well as on the intrinsic structure of a DE scenario.

CCS Concepts

•Information systems → Data exchange;

1. INTRODUCTION

Over the last decade, a plethora of mapping systems, including commercial ones such as IBM Rational Data Architect and research prototypes [1], have been developed for data transformation and data integration tasks. Data Exchange (DE) is one of the core processes of data transformation, relying on first-order logic and as such mainly pursued in research implementations. It revolves around translating data adhering to a source schema into data compliant with a target schema, and satisfying a set of logic-based constraints. These constraints typically include: source-to-target (s-t) tuple-generating dependencies (tgds) and target constraints such as target tgds and target equality-generating dependencies (egds). Target egds in turn include primary keys, and more general functional dependencies (fds) on the target schema. The produced solution of a DE process, called target solution, is generally obtained using the chase algorithm. Such algorithm iteratively applies s-t and target constraints until a fix point (i.e termination) is reached; the chase result upon termination then yields the target solution.

Existing DE engines span from completely covering all the above classes of constraints to supporting only subsets thereof. Indeed, custom chase engines [3] have been conceived for computing DE solutions under a wide range of constraints. While such engines may show high efficiency when dealing with tgds and other complex constraints, target fds yet hinder their performance and scalability. Alternatively, to aim for performance, DE engines like [4] have focused on outputting a set of SQL queries whose execution yields the target solution. While fast indeed, this approach is, however, mostly limited to s-t constraints. Extensions to subsets of target fds were shown possible, but typically requiring the additional input of source constraints [4].

Contributions. We demonstrate ChaseFUN, a novel chase-based engine for Data Exchange in the presence of arbitrary target fds and in the absence of source constraints. Our demonstration's first focus will be on emphasizing our system's *performance* on such DE scenarios. Indeed, as we will show, ChaseFUN is able to dramatically speed-up fd evaluation by leveraging constraints' interaction and chase step ordering, and exploiting the granular processing and parallelization opportunities yielded by such concepts. By showcasing our system's performance and scalability on large and complex DE scenarios, we then aim at showing that efficient support is yet attainable for general target fds, despite the overhead brought in by these constraints.

Interestingly, the concepts that stand at the core of ChaseFUN's performance endorse our system with an additional property: the ability of shedding light on the internals of DE scenarios and the corresponding chase sequences. To this end, ChaseFUN offers several features allowing the user to consult and examine scenario and chase-related data, including a particularly informative step-by-step execution of the chase procedure. Accordingly, our demonstration's second focus will be on showcasing such features, thus *providing the user with a novel, behind-the-scenes view on the underpinnings of DE.* To the best of our knowledge, such view has never been previosuly proposed by a chase-based DE engine.

Paper layout. We present an overview of ChaseFUN in Section 2 and the demonstration details in Section 3.

Series ISSN: 2367-2005 10.5441/002/edbt.2017.63

(i) Source Instance I

Active_Actors

name	*surname*	*age*
Leonardo	Di Caprio	40
John	Redmayne	33

Actor_Collaboration

name$_1$	*surname*$_1$	*name*$_2$	*surname*$_2$
Leonardo	Di Caprio	Matthew	David
Fredric	March	Miriam	Hopkins

Awarded_Actor

name	*surname*	*oscarName*	*year*
John	Redmayne	Best Actor	2014
Wallace	Beery	Best Actor	1932
Fredric	March	Best Actor	1932
Marlon	Brando Jr	Best Actor	1954
Marlon	Brando Jr	Best Actor	1972

(ii) Dependencies (uppercase for existential variables)

$\mathbf{m_1}$:
$Active\_Actors(n, s, a) \rightarrow Actor(n, s, Y_1, Y_2)$

$\mathbf{m_2}$:
$Awarded\_Actor(n', s', p', w') \rightarrow$
$\quad Actor(n', s', T, T_1) \wedge Oscar\_Prize(p', w', T)$

$\mathbf{m_3}$:
$Actor\_Collaboration(n'', s'', n''', s''') \rightarrow$
$\quad Actor(n'', s'', E_1, E_2) \wedge Actor(n''', s''', E_3, E_2)$

$\mathbf{e_1}$:
$Actor(n, s, p, w) \wedge Actor(n, s, p', w') \rightarrow$
$\quad (p = p') \wedge (w = w')$

$\mathbf{e_2}$:
$Oscar\_Prize(p, w, z) \wedge Oscar\_Prize(p, w, z') \rightarrow (z = z')$

(iii) Target Instance (Solution) J
(values N_x are labelled nulls)

Actor

*name**	*surname**	*idRewarding*	*idClub*
John	Redmayne	N$_5$	N$_6$
Wallace	Beery	N$_7$	N$_8$
Marlon	Brando Jr	N$_{13}$	N$_{14}$
Leonardo	Di Caprio	N$_{15}$	N$_{16}$
Matthew	David	N$_{17}$	N$_{16}$
Fredric	March	N$_7$	N$_{19}$
Miriam	Hopkins	N$_{20}$	N$_{19}$

Oscar_Prize

*oscarName**	*year**	*idActor*
Best Actor	2014	N$_5$
Best Actor	1932	N$_7$
Best Actor	1954	N$_{13}$
Best Actor	1972	N$_{13}$

Figure 1: Running example: DE scenario involving actors, prizes and collaborations.

$\mathbf{m_1}$
$a_1m_1 = \quad \{n : Leonardo,\ s : Di\ Caprio,\ a : 40,\ Y_1 : N_1,\ Y_2 : N_2\}$
$a_2m_1 = \quad \{n : John,\ s : Redmayne,\ a : 33,\ Y_1 : N_3,\ Y_2 : N_4\}$

$\mathbf{m_2}$
$a_1m_2 = \quad \{n' : John,\ s : Redmayne,\ p' : BestActor,\ w' : 2014,\ T : N_5,\ T_1 : N_6\}$
$a_2m_2 = \quad \{n' : Wallace,\ s : Beery,\ p' : BestActor,\ w' : 1932,\ T : N_7,\ T_1 : N_8\}$
$a_3m_2 = \quad \{n' : Fredric,\ s : March,\ p' : BestActor,\ w' : 1932,\ T : N_9,\ T_1 : N_{10}\}$
$a_4m_2 = \quad \{n' : Marlon,\ s : Brando\ Jr,\ p' : BestActor,\ w' : 1954,\ T : N_{11},$
$\quad T_1 : N_{12}\}$
$a_5m_2 = \quad \{n' : Marlon,\ s : Brando\ Jr,\ p' : BestActor,\ w' : 1972,\ T : N_{13},$
$\quad T_1 : N_{14}\}$

$\mathbf{m_3}$
$a_1m_3 = \quad \{n'' : Leonardo,\ s'' : Di\ Caprio,\ n''' : Matthew,\ s''' : David,$
$\quad E_1 : N_{15},\ E_2 : N_{16},\ E_3 : N_{17}\}$
$a_2m_3 = \quad \{n'' : Fredric,\ s'' : March,\ n''' : Miriam,\ s''' : Hopkins,$
$\quad E_1 : N_{18},\ E_2 : N_{19},\ E_3 : N_{20}\}$

(i) Set of assignments in their initial form (values N_x are labelled nulls)

$S_1 = \quad \{a_1m_1,\ a_1m_3\}$
$S_2 = \quad \{a_2m_1,\ a_1m_2\}$
$S_3 = \quad \{a_2m_2,\ a_3m_2,\ a_2m_3\}$
$S_4 = \quad \{a_4m_2,\ a_5m_2\}$

(ii) Saturation Sets

$a_1m_1 = \quad \{n : Leonardo,\ s : Di\ Caprio,$
$\quad a : 40,\ Y_1 : N_{15},\ Y_2 : N_{16}\}$
$a_1m_3 = \quad \{n'' : Leonardo,\ s'' : Di\ Caprio,$
$\quad n''' : Matthew,\ s''' : David,$
$\quad E_1 : N_{15},\ E_2 : N_{16},\ E_3 : N_{17}\}$

(iii) S_1 after chase

Actor:			
Leonardo	Di Caprio	N_{15}	N_{16}
Matthew	David	N_{17}	N_{16}

(iv) Materialization of S_1 after chase

Figure 2: Assignments and Saturation Sets for the DE scenario in Figure 1.

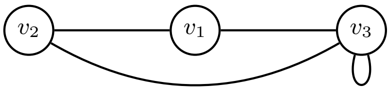
2. SYSTEM OVERVIEW

Main algorithmic concepts. To efficiently produce DE solutions, ChaseFUN relies on a series of algorithmic concepts which we synthetically illustrate hereafter[1] by means of a DE example depicted in Figure 1: Figure 1(i) shows the source instance I; (ii) shows the s-t tgds (m_1, m_2, m_3) and target fds (e_1 and e_2); finally, (iii) shows the target solution J. This example shows a recurring transformation task, that of taking overlapping data across source tables (e.g. the actors who are active, who collaborate with each other, and win prizes) and injecting them into one or two target tables by merging duplicates via target functional dependencies. Transformations of this kind, involving general target fds and no source constraints, are indeed crucial in DE. Using our example, we describe hereafter the key concepts and tools used by ChaseFUN:

- Chase and assignments. Our chase flavor relies on the construction, selection and modification of a set of full s-t tgd assignments corresponding to the DE scenario. Each assignment is initially a mapping of universal variables in the s-t tgd body to source constants, further enriched with a mapping of existential variables in the s-t tgd head to fresh labelled nulls. Initial assignments for our running example are illustrated in Figure 2(i). Chase steps with s-t tgds consist in the selection of a yet available assignment, which is marked as no longer available and added to a target set. Chase steps with egds (fds) in turn modify assignments within the current target set. Upon termination of the chase, the target set comprises the final (i.e. potentially modified by egd application) form of all assignments. We obtain the tuples in the target instance J by *materializing* this final form, i.e. by replacing variables in the tgds heads with their assigned values.
- Saturation Sets and chase order. One of the main reasons behind ChaseFUN's performance is its ability to tame the size of the intermediate target set during the chase, thus *systematically reducing the egd application scope.* To achieve such reduced size, we *group* assignments that are estimated to be at some point interacting via fds. We call such groups of assignments *Saturation Sets.* The chase of a Saturation Set will typically alternate between tgd steps and series of egd steps, applied to termination (i.e., until no egd remains applicable). Each Saturation Set thus acts as an *independent chase unit* that provides a part of the target solution. Figure 2(ii) shows a possible partition of the assignments in our running example into four Saturation Sets. We further show, in Figure 2(iii), the result of chasing S_1 (two s-t tgd steps corresponding to the assignments' selection, followed by an egd step with e_1 that modifies a_1m_1). Materializing this chase result yields the *Actor* tuples in Figure 2(iv). Note that these are indeed part of the target instance J in Figure 1(iii).
- The Conflict Graph and parallelization. To efficiently build Saturation Sets, our system uses a statically-built data structure called the *Conflict Graph.* Conflict Graph nodes correspond to s-t tgds, whereas edges witness the fact that the two s-t tgds, representing the connected nodes, have assignments that should potentially belong together in the

[1] A detailed description of these concepts is available in [2].

same Saturation Set. We call this kind of relation a *conflict* between two s-t tgds. The Conflict Graph further characterizes, via *conflict areas* adorning vertices, the interaction we would expect between assignments of the respective s-t tgds. The Conflict Graph for our running example is depicted below, with vertices v_1, v_2, v_3 corresponding to s-t tgds m_1, m_2, m_3.



$$\mathbf{Areas(v_1)} = \{ca_1^1 = \langle (n,s), e_1 \rangle\}.$$
$$\mathbf{Areas(v_2)} = \{ca_2^1 = \langle (n',s'), e_1 \rangle, ca_2^2 = \langle (p',w'), e_2 \rangle\}.$$
$$\mathbf{Areas(v_3)} = \{ca_3^1 = \langle (n'',s''), e_1 \rangle, ca_3^2 = \langle (n''',s'''), e_1 \rangle\}.$$

By v_1 and v_2's adornments we infer that any assignments of m_1 and m_2 may trigger the fd e_1, if they agree on the values for n and n', respectively s and s'. Thus, since they exhibit such agreement, a_2m_1 and a_1m_2 must belong together in the same Saturation Set, i.e. S_2 in Figure 2(ii).

Besides its important role in Saturation Set construction, the Conflict Graph also provides very interesting *parallelization* opportunities. Indeed, one can show that a Saturation Set can never span across several connected components of the graph. ChaseFUN thus proceeds to Saturation Set construction and chase *in parallel* for each of the Conflict Graph's connected components. Coupled to the Saturation Set-chase paradigm, parallel processing in turn further boosts our system's speed and scalability.

Implementation and assessment. We have implemented ChaseFUN in Java (JVM version 1.8) using a JDBC interface for communication with an underlying *PostgreSql9.4* DBMS system. To stress-test ChaseFUN we have used several scenarios generated by using iBench[1], a novel data integration benchmark for generating arbitrarily large and complex schemas and constraints. We have considered three types of scenarios, in increasing complexity order: (i) OF scenarios generated with the default iBench *object fusion* primitive; (ii) OF$^+$ scenarios, generated by combining the iBench *object fusion* and *vertical partitioning* primitives; (ii) OF^{++} scenarios, obtained by further modifying OF$^+$ to yield s-t tgds with up to three atoms in the head. To further provide scale and assess the signficance of ChaseFUN's performance, we comparatively ran, on the same scenarios, one of the best DE engines currently available, namely the Llunatic system[3]. Figure 3 shows several measures obtained during this comparative evaluation[2].

Scenarios

SCENARIO	*s-t tgds*	*OF*	*OF+*	*OF++*	*# source tuples*
A	15	5 egds	10 egds	15 egds	500K
C	45	15 egds	30 egds	45 egds	1.5M
F	90	30 egds	60 egds	90 egds	3M

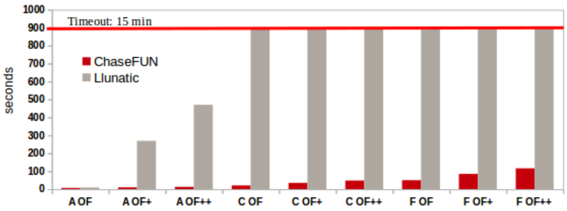


Figure 3: Evaluation and comparative assessment.

DE workflow. Our system runs the DE process as a transition among four states, detailed hereafter.

- 1. Initial state: waiting to load scenario. Prior to any interaction, ChaseFUN bootstraps with *loading a Data Exchange scenario*, comprising source and target schemas, constraints (s-t tgds and target fds) and source instance tuples.
- 2. Ready to chase state. Once a scenario has been loaded, the Conflict Graph and the initial assignments are further computed. The system then reaches the *Ready to chase* state, where the user can browse scenario-related data: source and target schemas, source instance, s-t tgds and their assignments, target fds, as well as the Conflict Graph.

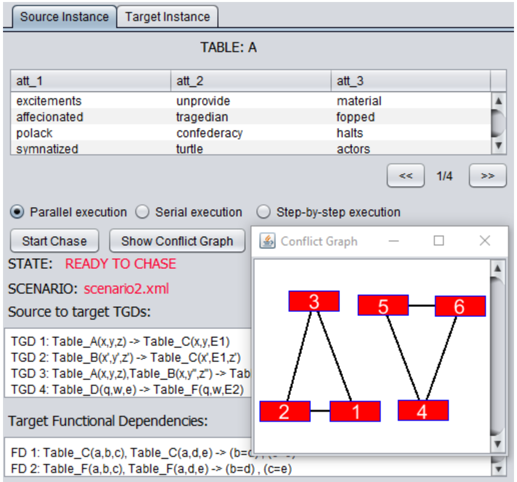


TABLE: A

att_1	att_2	att_3
excitements	unprovide	material
affecionated	tragedian	fopped
polack	confederacy	halts
symnatized	turtle	actors

● Parallel execution ○ Serial execution ○ Step-by-step execution

STATE: READY TO CHASE

SCENARIO: scenario2.xml

Source to target TGDs:

TGD 1: Table_A(x,y,z) -> Table_C(x,y,E1)
TGD 2: Table_B(x',y',z') -> Table_C(x',E1,z')
TGD 3: Table_A(x,y,z),Table_B(x,y'',z'') -> Tab
TGD 4: Table_D(q,w,e) -> Table_F(q,w,E2)

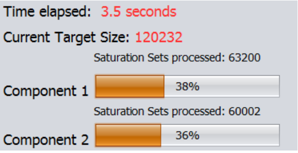
Target Functional Dependencies:

FD 1: Table_C(a,b,c), Table_C(a,d,e) -> (b=c
FD 2: Table_F(a,b,c), Table_F(a,d,e) -> (b=d) , (c=e)

Figure 4: Ready to chase state for Scenario 2.

ChaseFUN provides windows and subwindows where baseline information can be selectively displayed by clicking on the corresponding tabs. Details can be further obtained by clicking on displayed elements. Figure 4 shows some of the system's visual feedback in the Ready to chase state for our demonstration Scenario 2.

- 3. Chase in progress state. Pressing the *Start Chase* button triggers the start of the chase procedure, with a choice among three *chase modes*. The first two modes both imply a continuous run, corresponding to a serial (sequential) and respectively parallel processing of the connected components in the Conflict Graph. The third mode in turn is aimed at allowing the user to *peak into the chase*, via a step-by-step execution. We detail this mode at the end of this section.



Time elapsed: 3.5 seconds

Current Target Size: 120232

Saturation Sets processed: 63200

Component 1

Saturation Sets processed: 60002

Component 2

Figure 5: Progress information for Scenario 2.

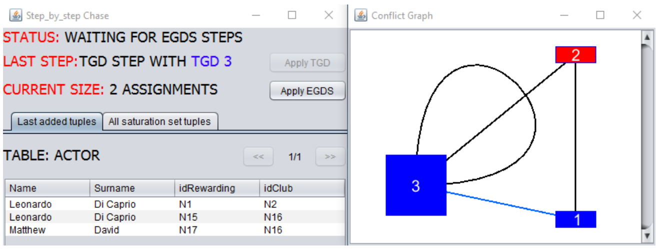
Throughout the chase, our system displays a range of useful information regarding the current state and evolution of the chase. This comprises progress bars for each connected component, the time spent chasing so far, as well as the evolving size of the solution so far constructed, by progressive materialization of completed Saturation Sets. Figure 5 illustrates such progress-related information.

- 4. Final state: chase completed. Upon chase completion, in addition to previously available information, the user has access to the contents of the solution, as well as to a wide range of time and size statistics. She may export these

[2] We ran experiments on a 4-cores, i7-6600U 2.6 Ghz, 8GB RAM machine. We set a 15min timeout for all runs. We used the latest, most optimized version of Llunatic, as provided by its authors.

statistics and/or wraparound to the initial state to run ChaseFUN on a new DE scenario.

Step-by-step chase. An essential feature of ChaseFUN is that of providing a detail-oriented, debug-like, step-by-step chase mode, aimed towards learning and understanding *how the chase goes* and what ChaseFUN's unit actions are. When the step-by-step option is selected, a new window pops up, allowing the user to incrementally run and inspect the results of each Saturation Set's construction and chase, alternating between tgds and egds. To improve the understanding of this process, ChaseFUN will provide a range of additional status information and visual cues.



STATUS: WAITING FOR EGDS STEPS

LAST STEP:TGD STEP WITH TGD 3

CURRENT SIZE: 2 ASSIGNMENTS

TABLE: ACTOR

Name	Surname	idRewarding	idClub
Leonardo	Di Caprio	N1	N2
Leonardo	Di Caprio	N15	N16
Matthew	David	N17	N16

Figure 6: Step-by-step chase for Scenario 1

Figure 6 shows a snapshot of the step-by-step chase for our demonstrated Scenario 1. This scenario corresponds to our running example in Figure 1 and we refer the reader to the detailed description of this example above. The snapshot corresponds to the construction and chase of the Saturation Set S_1. In particular, it depicts the state reached after the addition of the assignment a_1m_3 to S_1. The user has thus previously launched two tgd steps, namely for m_1 and m_3, whose corresponding Conflict Graph nodes have accordingly changed colour. Furthermore, the last tgd to add an assignment being m_3, its corresponding node is emphasized (enlarged). The edge linking m_1 and m_3 is equally emphasized (shown in blue), since a_1m_3 has been added because of its estimated interaction with an assignment of m_1 (i.e. a_1m_1). A subwindow displays the tuples obtained by the materialization of the current Saturation Set. Since after each tgd step egds must be applied, this is signaled to the user via the status information and the available button. Expectedly, once the user launches the next egds step, the tuples shown in Figure 6 will evolve to become the tuples shown in Figure 2(iv).

The step-by-step chase is importantly made available by our system's "by design" granular processing of the chase, keeping the user-intended information small enough to remain easily accessible and understandable. To account for large-sized scenarios, ChaseFUN additionally provides *pause/ continue*-like interactions, by letting the user alternate between the continuous serial and the step-by-step mode over the course of a single chase sequence.

3. DEMONSTRATION OVERVIEW

Scenarios. We will demonstrate our system on scenarios of increasing complexity in terms of both the number of constraints and the source instance size, namely one synthetic and three iBench-based[1] DE scenarios detailed hereafter.

- Scenario 1 is our simplest scenario, corresponding to our running example in Figure 1 and comprising 9 tuples in the source, 3 s-t tgds, 2 egds and a single connected component in the Conflict Graph.
- Scenario 2 is on the mid-low side of the complexity spectrum. It comprises $400K$ tuples in the source and is built using twice the iBench default *object fusion* primitive (see Section 2), yielding 6 s-t tgds, 2 egds, and 2 connected components in the Conflict Graph.
- Scenario 3 increases the source instance size to $1M$ tuples, and further raises complexity by (i) increasing the number of iBench *object fusion* primitives applied and (ii) further plugging-in the *vertical partitioning* iBench primitive (in terms of Section 2 notation, this is an OF^{+} scenario). It includes 30 s-t tgds, 30 egds, and 10 connected components in the Conflict Graph.
- Scenario 4 raises the bar to $3M$ tuples in the source, and a larger yet number of constraints: 90 s-t tgds and 90 egds, yielding a Conflict Graph of 30 connected components. We obtain this scenario by plugging in both *object fusion* and *vertical partitioning* primitives and further increasing the number of atoms in the s-t tgds heads. Scenario 4 is in fact our OF^{++} stress-test scenario F in Figure 3.

Showcased features and messages conveyed. On the above scenarios, we will demonstrate our system's features and interactions described in Section 2, emphasizing ChaseFUN's two main strengths:

- Performance. We will showcase our system's processing speed and ability to scale for large and complex Data Exchange scenarios with target fds. As also witnessed by our experimental assessment, we are indeed not aware of a previous DE engine able to equate or outperform ChaseFUN in such settings. Since parallelization is one of our key performance factors, we will moreover show its impact and benefits by providing comparative runs using the parallel and serial chase modes offered by ChaseFUN. To present performance results, we will in particular focus on Scenarios 3 and 4. We also offer the possibility of live running comparative assessments of our system, such as the one charted in Section 2.
- User-intended view on the DE internals. We will showcase the available Conflict Graph metadata, enabling a global, synthetic view on the links and interplay of constraints in the demonstrated DE scenarios. We will further emphasize the usefulness of the chase progress information provided by our system, as a first and important solution against the opacity problem of the chase operated by DE engines. Finally, we will extensively present the step-by-step chase mode described in Section 2, aimed at offering a novel, behind-the-scenes, refined view of the "low-level" granular operations of the DE process. We will showcase these capabilities on all demonstrated scenarios, and use Scenario 1 for an end-to-end presentation of the step-by-step run. Our demonstration will particularly focus on these detail and introspection opportunities provided by ChaseFUN. Indeed, to the best of our knowledge, ours is the first DE engine to provide the users with such informative and instructive features.

4. REFERENCES

[1] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. The ibench integration metadata generator. *PVLDB*, 9(3):108–119, 2015.

[2] A. Bonifati, I. Ileana, and M. Linardi. Functional dependencies unleashed for scalable data exchange. In *Proceedings of SSDBM*, pages 2:1–2:12, 2016.

[3] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and cleaning. In *Proceedings of ICDE*, p. 232–243, 2014.

[4] B. Marnette, G. Mecca, P. Papotti, S. Raunich, and D. Santoro. ++Spicy: an OpenSource Tool for Second-Generation Schema Mapping and Data Exchange. *PVLDB*, 4(12):1438–1441, 2011.

GnosisMiner: Reading Order Recommendations over Document Collections

Georgia Koutrika
HP Labs
Palo Alto, CA, USA

Alkis Simitsis
Hewlett Packard Labs
Palo Alto, CA, USA

Yannis Ioannidis
University of Athens and
"Athena" Research Center
Athens, Greece

ABSTRACT

Given a document collection, existing systems allow users to locate documents either using search keywords or by navigating through some predefined organization of the collection. Other approaches help the user understand a collection by generating summaries or clusters of the documents at hand. However, often users would like to understand how the documents may be related to each other and access them in some logical order. In this work, we present an interactive reading recommendation system, called *GnosisMiner*. Given a collection of documents and a theme, the system returns a partial order of documents relevant to that theme organized from more general to more specific. The recommended *reading order* resembles the human approach of learning as we typically start our path to knowledge from more general documents that help us understand the domain and then we proceed with more specific, more specialized documents to increase our knowledge of the matter.

1. INTRODUCTION

Given a document collection, existing systems allow users to locate documents either using search keywords or by navigating through some predefined organization of the collection. However, search engines hide document relationships, and navigational interfaces capture only fixed relationships that do not dynamically adapt to the users' specific needs. Therefore, while these systems work fine when users try to locate specific documents, they are insufficient when users would like to understand how the documents may be related to each other and access them in some logical order.

We advocate that given a document collection, it is very useful to recommend to a user a possible *reading order* over this collection so that she can access the documents in an organized, structured way. In the past, there have been efforts towards helping a user understand and access a corpus of documents in some meaningful way. These efforts include corpus summarization approaches, which try to generate a textual summary of the collection [9, 10], hierarchical document clustering methods, which segment the corpus [6, 7, 11], and document linking, which connect documents through specific types of links such as 'consequence of' or 'follow-up' [2, 8]. Google's advanced search interface [3] organizes search results into three reading levels: basic, intermediate, and advanced, offering a very coarse document ordering.

In this work, we present an interactive reading recommendation system, called *GnosisMiner*. Given a collection of documents and given a theme (i.e., a set of keywords), our system returns a partial order of documents relevant to that theme organized from more general to more specific. The recommended *reading order* resembles the human approach of learning as we typically start our path to knowledge from more general documents that help us understand the domain and then we proceed with more specific, more specialized documents to increase our knowledge of the matter.

GnosisMiner represents a reading order as a tree and users may select which path on the tree they would like to follow and, hence, which documents they would like to read in order. To help them further, the system shows the topics found in each document as well at each level of the tree. Users may modify the recommended reading orders through parameters that determine how fine-grained the ordering should be. Finally, the system provides several visualizations of the underlying collection aimed at the expert user who would like to gain insights into the similarities and topics of the documents and tune the recommendations accordingly.

Recommending reading orders is useful in many areas, such as (*a*) education, for organizing online educational material, (*b*) patent searching, for helping users (e.g. patent attorneys) to understand which patents have more general cover than others, (*c*) research, for organizing publications or news articles to help researchers study a topic, (*d*) publishing, for helping editors select and organize articles to publish on a web site, and so forth.

2. RECOMMENDING READING ORDERS

Given a collection of documents, a reading order is a partial order of the documents from general to more specific documents. In this partial order, there are two types of document relationships, equivalence and precedence, which can be informally described as follows. If two documents are about the same topics, then they are considered *equivalent*, denoted $a \leftrightarrow b$, and are grouped together. If they are about related topics but document b is more specific than a, then a *precedes* b in the order, denoted $a \rightarrow b$.

As an example, consider the following documents: a is an introduction to data mining, b is on classification methods, and c is another introductory document on data mining. Both a and c cover the same topic to a similar extent and hence they are considered equivalent. Consequently, one can choose to read any of them. However, b is more focused, hence it has precedence relationships to the other documents: $a \rightarrow b$ and $c \rightarrow b$.

We quantitatively define the equivalence and precedence relationships in a reading order using two metrics: *document generality* and *document overlap*. We will first describe the metrics and then show how the two relationships are defined with their help.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.64

Given documents a and b, the document generality for a document a is captured by the *generality score*, $g(a)$, a real number such that higher values mean higher generality. In other words, a is more general than b iff $g(a) \geq g(b)$. The document overlap for the pair a and b is captured by the *overlap score*, $o(a, b)$, a real number typically in the range of $[0, 1]$, where 0 means no overlap between a and b, and 1 means maximum overlap.

We measure document overlap and generality based on the documents' topical relationships. To derive the topics describing the documents, we use topic modeling. Topic models [1] are based upon the idea that documents are mixtures of topics, where a topic is a probability distribution over words. A topic model aims at discovering the hidden thematic structure of a collection of documents by finding how topics are assigned to documents, and how topics are described by words in the documents. Representing a document using topics rather than document keywords is more effective because it allows capturing implicit relationships between documents, not just the explicit similarity of their common words.

Document generality. We compute the document generality as a measure of the document's entropy over the topics it covers. The basic intuition behind the entropy is that the higher a document's entropy is, the more topics it covers in less depth hence the more general it is. Given a collection $\mathcal{D}$ of n documents and s topics, we denote $F_{n\times s}$ the document-topic matrix that captures how the s topics are assigned to the n documents in $\mathcal{D}$. $F_{ij} \in [0, 1]$ with $i \leq n$ and $j \leq s$ describes how well topic t_j describes document a_i. Using the Shannon entropy, the *generality score* $g(a_i)$ of document a_i can be defined as follows:

$$g(a_i) = H(a_i) = \sum_j -F_{ij} \log(F_{ij}) \tag{1}$$

Document overlap. The topic overlap $o(a, b)$ of two documents a and b can be defined using the weighted Jaccard score [4]. The weighted Jaccard extends the classic Jaccard index, which is defined as the size of the intersection divided by the size of the union of the topic sets assigned to each document, by taking into account how well a topic represents a document. The topic overlap can be defined as follows:

$$o(a, b) = Jaccard(a, b) = \frac{F_a \cdot F_b}{|F_a|^2 + |F_b|^2 - F_a \cdot F_b} \tag{2}$$

where F_a (F_b) is the topic vector associated with a (b, resp.). Larger values indicate more common topics between two documents.

Note that other metrics for measuring document generality and overlap are possible. For example, instead of the Shannon entropy, we could use the residual entropy (entropy of non-common terms) or the distribution entropy (entropy of the location of common, non-common, or both types of terms throughout the document).

Now we can formally define the document equivalence and precedence relationships in a reading order as follows:

$$\textit{document equivalence: } a \leftrightarrow b \text{ iff } |g(a) - g(b)| \leq \kappa \wedge o(a, b) \geq \tau \tag{3}$$

$$\begin{aligned} \textit{document precedence: } a \rightarrow b \text{ iff } g(a) > g(b) \wedge o(a, b) > 0 \\ \wedge \ (|g(a) - g(b)| > \kappa \ \vee \ o(a, b) < \tau) \end{aligned} \tag{4}$$

τ defines the minimum topic overlap between two equivalent documents and κ defines the maximum difference of their generality scores.

Figure 1(a) shows an example reading order over six documents. A user can follow different reading paths following the document relationships, such as the example reading path: $d_1 \rightarrow d_4 \rightarrow d_6$, shown in the figure. Furthermore, there may be more than one reading orders for the same set of documents. For example, consider documents d_1, d_2 and d_3, which have some overlap and $d_1 \leftrightarrow d_2$ and $d_2 \leftrightarrow d_3$ but d_1 is not equivalent to d_3. Figure 1(b) shows two possible reading orders.

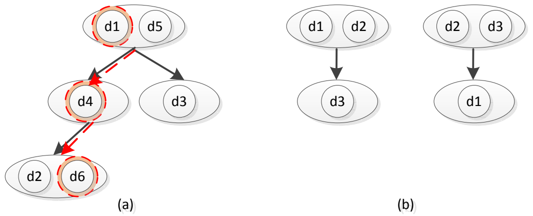


Figure 1: Example reading orders

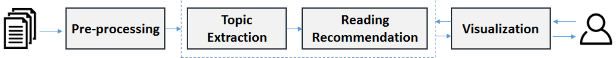


Figure 2: System architecture

3. SYSTEM OVERVIEW

We present GnosisMiner, a prototype system for recommending ordered readings over document collections. The system architecture is depicted in Figure 2. Its main components are: *pre-processing*, *topic extraction*, *reading recommendation*, and *visualization*. The users interact with the system through the visualization component to specify the document collection and the theme of their interest, interact with the recommended reading order and the documents, modify the recommendation parameters, and examine the various visualizations over the collection.

Next, we describe the main components of our system. For more details on the algorithms used for topic extraction and reading order recommendation we refer the interested reader to [5].

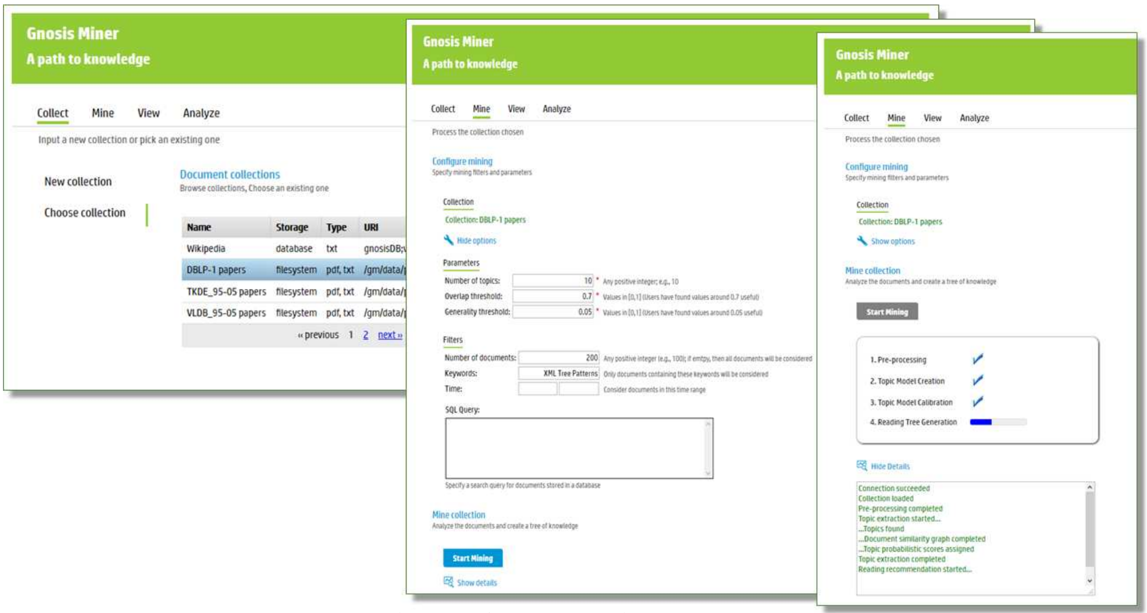
3.1 Pre-processing

Pre-processing removes noisy and stop words, performs stemming, and transforms each document to a term vector using a tf-idf weighting scheme. We perform this task incrementally; we skip documents already processed in a previous run and only work on documents never processed before.

3.2 Topic extraction

To measure the generality and overlap of the documents, and identify the topical relationships among them, we first derive the topics that describe the documents. The *topic extraction module* works in two phases. First, it extracts the topics that occur in the documents using the Latent Dirichlet Allocation (LDA) model with Gibbs sampling [1]. However, topic models often misassign or miss topics for documents. To reduce such errors, subsequently, the topic extraction uses a score propagation method that allows the topic scores of a document to be influenced by the topics of its most similar neighbors. This module leverages the content similarity of the documents by comparing their term representations and propagates document-topic scores between strongly similar documents on the basis that due to their similarity they likely have similar topics.

For this purpose, this module first builds the document similarity graph, where each node maps to a document, and each edge between two documents captures their similarity (i.e., the similarity of their term-based representations) in the edge weight. Then, the document-topic scores, returned by LDA during the first step of topic extraction, are propagated over the document similarity graph, so the potential topics of a document take into consideration the topic scores of their neighbors (which in turn, depend on the scores of their respective neighbors, and so on). The algorithm iteratively updates the topic scores of a node (document) based on the weighted average of the scores of its neighbors.



Input a new collection or pick an existing one

Document collections
Browse collections, Choose an existing one

Name	Storage	Type	URI
Wikipedia	database	txt	gnosisDB;\
DBLP-1 papers	filesystem	pdf, txt	/gm/data/
TKDE_95-05 papers	filesystem	pdf, txt	/gm/data/
VLDB_95-05 papers	filesystem	pdf, txt	/gm/data/

Process the collection chosen

Configure mining
Specify mining filters and parameters

Collection
Collection: DBLP-1 papers

Parameters
Number of topics: 10 * Any positive integer; e.g., 10
Overlap threshold: 0.7 * Values in [0,1] (Users have found values around 0.7 useful)
Generality threshold: 0.05 * Values in [0,1] (Users have found values around 0.05 useful)

Filters
Number of documents: 200 Any positive integer (e.g., 100); if emtpy, then all documents will be considered
Keywords: XML Tree Patterns Only documents containing these keywords will be considered
Time: Consider documents in this time range

SQL Query:
Specify a search query for documents stored in a database

Mine collection
Analyze the documents and create a tree of knowledge

1. Pre-processing ✓
2. Topic Model Creation ✓
3. Topic Model Calibration ✓
4. Reading Tree Generation

```
Connection succeeded
Collection loaded
Pre-processing completed
Topic extraction started...
...Topics found
...Document similarity graph completed
...Topic probabilistic scores assigned
Topic extraction completed
Reading recommendation started...
```

Figure 3: Collect and Mine components

3.3 Reading Recommendation

The *reading recommendation* uses the document-topic scores learnt from the topic extraction module to determine the equivalence and precedence relations among the documents and mine a reading order for them. This module takes as input a set of documents, the document-topic assignments, and the parameters τ, which defines the minimum topic overlap between two equivalent documents, and κ, which defines the maximum difference of their generality scores. The module uses an iterative method to build a tree that represents the recommended reading order.

In this tree, nodes correspond to the input documents and edges capture precedence relationships between the documents. In particular, a node maps a non-empty set of equivalent documents. An edge between nodes A and B signifies that documents belonging to the corresponding document set of A precede the documents belonging to the respective node B.

For the root of this tree, the method puts together the most general, equivalent documents (i.e, documents whose generality difference is small ($< \kappa$) and whose overlap is high ($> \tau$)). From the remaining documents, the method creates clusters of documents that can be grouped together because they overlap with each other and they also have some overlap ($0 <$ and $< \tau$) with the root of the tree. Each of these clusters will be used to grow a subtree that will be connected to the current node (let's call it the parent node of the cluster) in subsequent rounds. The reading recommendation module takes each of the clusters created in the previous cycle and selects the most general, equivalent documents. This set becomes a new node that is added under the parent node of the cluster. Then, the remaining documents are clustered. Note that in each clustering step, documents that were un-clustered before may get grouped now. This process repeats until no more tree growing is possible and there are no documents unprocessed.

3.4 Visualization

The *visualization component* allows the user to interact with the system. The user can specify the document collection they would like to explore and the theme of their interest (e.g., as a set of keywords), interact with the recommended reading order of the documents, modify the recommendation parameters, and examine various visualizations over the collection.

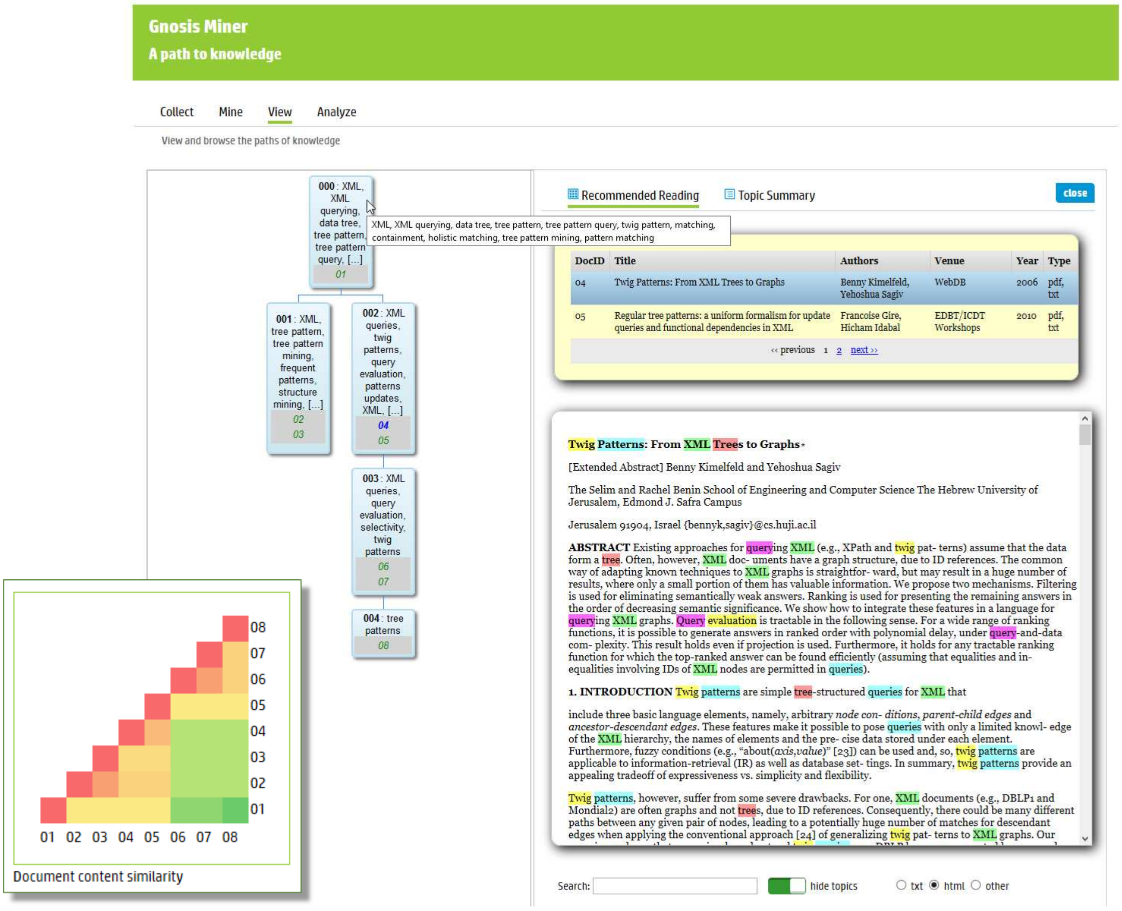
GnosisMiner visualizes a reading order as a tree, where each node corresponds to a set of equivalent documents. The system shows the topics found in each document as well as at each node of the tree. The user can interact with the tree in a table-of-contents manner, and choose which documents to read in the proposed order. The user can modify the recommended reading order through parameters that determine how fine-grained the ordering should be. These parameters include the number of topics to use for describing the documents of interest, the minimum topic overlap (τ) between equivalent documents, and the maximum difference (κ) of their generality scores.

Finally, the system provides several visualizations of the underlying collection that offer a look under the hood at the document relationships as well as at the operation and performance of the system. The user can visually examine the topics describing the selected set of documents, how these topics are assigned to documents, the document content similarities, and their generality scores. For instance, the document content similarities are visualized using a heatmap. The user can also review details regarding the operation of the various components of the system, such as execution times, number of iterations of the topic extraction, number of iterations of the reading recommendation, and so forth.

4. OUR PRESENTATION

Our presentation will demonstrate GnosisMiner's features using a collection of data management related papers as our corpus. Our demonstration script starts with a small number of representative examples. With these examples we will show how a user can choose a collection and specify which part of the collection she is interested in. For example, Figure 3 shows an example navigation and run of the system, using a collection of papers, example parameters for topic extraction, and an example filter limiting the search to 200 papers with a theme 'XML Tree Patterns'. When a reading order has been generated, it is shown in the View component.

Figure 4 shows a snapshot of an abridged result for this example. The left panel contains the tree representing the reading order chosen. Each node contains a set of topics along with links to papers related to those topics. Hovering over a node shows the complete


Gnosis Miner
A path to knowledge
Collect
Mine
View
Analyze
View and browse the paths of knowledge
000 : XML, XML querying, data tree, tree pattern, tree pattern query, [...]
01
XML, XML querying, data tree, tree pattern, tree pattern query, twig pattern, matching, containment, holistic matching, tree pattern mining, pattern matching
001 : XML, tree pattern, tree pattern mining, frequent patterns, structure mining, [...]
02
03
002 : XML queries, twig patterns, query evaluation, patterns updates, XML, [...]
04
05
003 : XML queries, query evaluation, selectivity, twig patterns
06
07
004 : tree patterns
08
Recommended Reading
Topic Summary
close
08
07
06
05
04
03
02
01
01 02 03 04 05 06 07 08
Document content similarity
Search:
hide topics
○ txt ● html ○ other

DocID	Title	Authors	Venue	Year	Type
04	Twig Patterns: From XML Trees to Graphs	Benny Kimelfeld, Yehoshua Sagiv	WebDB	2006	pdf, txt
05	Regular tree patterns: a uniform formalism for update queries and functional dependencies in XML	Francoise Gire, Hicham Idabal	EDBT/ICDT Workshops	2010	pdf, txt

‹‹ previous 1 2 next ››

Twig Patterns: From XML Trees to Graphs*

[Extended Abstract] Benny Kimelfeld and Yehoshua Sagiv

The Selim and Rachel Benin School of Engineering and Computer Science The Hebrew University of Jerusalem, Edmond J. Safra Campus

Jerusalem 91904, Israel {bennyk,sagiv}@cs.huji.ac.il

ABSTRACT Existing approaches for querying XML (e.g., XPath and twig pat- terns) assume that the data form a tree. Often, however, XML doc- uments have a graph structure, due to ID references. The common way of adapting known techniques to XML graphs is straightfor- ward, but may result in a huge number of results, where only a small portion of them has valuable information. We propose two mechanisms. Filtering is used for eliminating semantically weak answers. Ranking is used for presenting the remaining answers in the order of decreasing semantic significance. We show how to integrate these features in a language for querying XML graphs. Query evaluation is tractable in the following sense. For a wide range of ranking functions, it is possible to generate answers in ranked order with polynomial delay, under query-and-data com- plexity. This result holds even if projection is used. Furthermore, it holds for any tractable ranking function for which the top-ranked answer can be found efficiently (assuming that equalities and in- equalities involving IDs of XML nodes are permitted in queries).

1. INTRODUCTION Twig patterns are simple tree-structured queries for XML that

include three basic language elements, namely, arbitrary *node con- ditions*, *parent-child edges* and *ancestor-descendant edges*. These features make it possible to pose queries with only a limited knowl- edge of the XML hierarchy, the names of elements and the pre- cise data stored under each element. Furthermore, fuzzy conditions (e.g., "about(*axis*,*value*)" [23]) can be used and, so, twig patterns are applicable to information-retrieval (IR) as well as database set- tings. In summary, twig patterns provide an appealing tradeoff of expressiveness vs. simplicity and flexibility.

Twig patterns, however, suffer from some severe drawbacks. For one, XML documents (e.g., DBLP1 and Mondial2) are often graphs and not trees, due to ID references. Consequently, there could be many different paths between any given pair of nodes, leading to a potentially huge number of matches for descendant edges when applying the conventional approach [24] of generalizing twig pat- terns to XML graphs. Our

Figure 4: View component (main picture) and example analysis chart: similarity heatmap (bottom-left corner)

list of topics of the node. In the figure, the root node of the tree contains the paper entitled "*M. Hachicha, J. Darmont: A Survey of XML Tree Patterns. IEEE Trans. Knowl. Data Eng. 25(1): 29-46 (2013)*", which is a survey paper on the specified subject matter, and it covers several topics in XML patterns. We observe that under this node, nodes *001* and *002* cover more focused topics: the former related to tree pattern mining, frequent patterns, structure mining, and so forth, and the latter related to XML queries, twig patterns, query evaluation, etc.

Selecting a paper link in a node opens the right panel, which shows the recommended list of papers for the corresponding set of topics and a summary listing of these topics. Choosing a paper from the list, shows the text in a viewer. For instance, in the figure, the document entitled "*Twig Patterns: From XML Trees to Graphs*" is viewed. A user can read the paper, perform a text search, and so on. There is also a show/hide topics feature that highlights the relevant topics in the text (enabled in the figure).

The advanced user may use the Analyze component to examine analysis charts (e.g., document-topics assignments, document similarities, performance statistics) to get insights into the document collection and refine the mining process if needed. The bottom-left corner of Figure 4 shows an example snapshot of a similarity heatmap for the 8 documents shown in the reading tree of Figure 4. For instance, one can see how documents *02 - 05* are closer in similarity to *01*, which is the root of the tree in the figure, while *06 - 08* are more distant. One could also see that a slightly more flexible similarity threshold could group documents *06 - 08* together.

Finally, for off-script presentation and discussion, we will provide interactivity, where the participants can explore the data set themselves and experiment with GnosisMiner.

5. REFERENCES

[1] David M. Blei. Probabilistic topic models. *Commun. ACM*, 55(4):77–84, April 2012.

[2] Ao Feng and James Allan. Incident threading for news passages. In CIKM, 2009.

[3] Google. Advanced search interface. Available at: http://www.google.ca/advancedsearch, 2016.

[4] G. Grefenstette. *Explorations in Automatic Thesaurus Discovery*. Kluwer Academic Publishers, 1994.

[5] Georgia Koutrika, Lei Liu, Steven J. Simske. Generating reading orders over document collections. ICDE, 507-518, 2015.

[6] Qirong Ho, Jacob Eisenstein, and Eric P. Xing. Document hierarchies from text and links. In WWW, 2012.

[7] Nachiketa Sahoo, Jamie Callan, Ramayya Krishnan, George Duncan, and Rema Padman. Incremental hierarchical clustering of text documents. In CIKM, 2006.

[8] Dafna Shahaf and Carlos Guestrin. Connecting two (or less) dots: Discovering structure in news articles. *ACM Trans. Knowl. Discov. Data*, 5(4):24:1–24:31, February 2012.

[9] B. Shaparenko and T. Joachims. Information genealogy: Uncovering the flow of ideas in non-hyperlinked document databases. In KDD, 2007.

[10] Ruben Sipos, Adith Swaminathan, Pannaga Shivaswamy, and Thorsten Joachims. Temporal corpus summarization using submodular word coverage. In CIKM, 2012.

[11] Ying Zhao, George Karypis, and Usama Fayyad. Hierarchical clustering algorithms for document datasets. *Data Min. Knowl. Discov.*, 10(2), March 2005.

MovieFinder: A Movie Search System via Graph Pattern Matching

Xin Wang [1,2] Chengye Yu [2] Enyang Zhang [2] Tong Du [2]

[1]*Southwest Jiaotong University* [2]*ChangHong Inc.*

xinwang@swjtu.cn, {chengye.yu, enyang.zhang, tong.du}@changhong.com

ABSTRACT

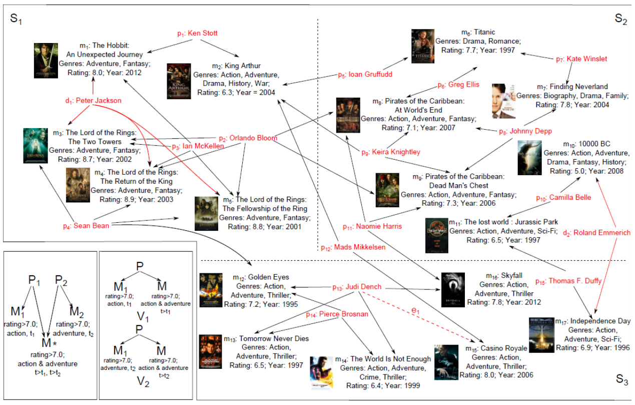
In this demo, we present MovieFinder, a user-friendly movie search system with following characteristics: it (1) searches movies on social networks via the technique of top-k graph pattern matching; (2) supports distributive computation to handle sheer size of real-life social networks; (3) applies view-based technique to optimize local evaluation, and employs incremental computation to keep cached views up to date; and (4) provides graphical interface to help users construct queries, explore data and inspect results.

1. INTRODUCTION

In recent years, social networking sites have experienced fast development, and are endowed with enormous commercial value. One key issue to achieve commercial goals via social networks is how to help uses find their interested objects on big social data. In light of this, a host of techniques are developed, among which graph pattern matching defined in terms of subgraph isomorphism has been widely used and verified to be effective [5].

However, it is nontrivial to efficiently conduct graph pattern matching on social networks due to the following reasons: (1) graph pattern matching with subgraph isomorphism is computationally expensive as it is an NP-complete problem [3], and moreover, there may exist exponentially many matches of a pattern query Q in a data graph G; (2) real-life graphs are typically large, *e.g.,* Facebook has 1.18 billion daily active users, and the average number of friends is 155 [1], it is hence prohibitively expensive to query such big graphs; (3) social networks are often distributively stored, which makes graph pattern matching more challenging or even infeasible; (4) social networks evolve constantly, it is often expensive to recompute matches starting from scratch when social networks are updated with minor changes.

Example 1: Consider a fraction of IMDb [2] collaboration network depicted as graph G in Fig. 1(a). Each node in G either denotes a performer (p) (resp. director (d)), labeled by *id*, *name*; or a movie (m), with attributes *title*, *genres* (g), *rating* (r) and *release time* (t). Each directed edge from a performer (resp. director) to a movie indicates that the performer (resp. director) played in (resp. directed) the movie, where the edges connecting directors and movies are marked in red. The graph G is *geo-distributed* to three sites S_1, S_2 and S_3, each storing a fragment of G.


S1
S2
S3
p1: Ken Stott
m1: The Hobbit: An Unexpected Journey Genres: Adventure, Fantasy; Rating: 8.0; Year: 2012
m2: King Arthur Genres: Action, Adventure, Drama, History, War; Rating: 6.3; Year = 2004
d1: Peter Jackson
m3: The Lord of the Rings: The Two Towers Genres: Adventure, Fantasy; Rating: 8.7; Year: 2002
p2: Orlando Bloom
p3: Ian McKellen
m4: The Lord of the Rings: The Return of the King Genres: Adventure, Fantasy; Rating: 8.9; Year: 2003
m5: The Lord of the Rings: The Fellowship of the Ring Genres: Adventure, Fantasy; Rating: 8.8; Year: 2001
p4: Sean Bean
m6: Titanic Genres: Drama, Romance; Rating: 7.7; Year: 1997
p7: Kate Winslet
p5: Ioan Gruffudd
p6: Greg Ellis
m7: Finding Neverland Genres: Biography, Drama, Family; Rating: 7.8; Year: 2004
m8: Pirates of the Caribbean: At World's End Genres: Action, Adventure, Fantasy; Rating: 7.1; Year: 2007
p8: Johnny Depp
p9: Keira Knightley
m10: 10000 BC Genres: Action, Adventure, Drama, Fantasy, History; Rating: 5.0; Year: 2008
m9: Pirates of the Caribbean: Dead Man's Chest Genres: Action, Adventure, Fantasy; Rating: 7.3; Year: 2006
p10: Camilla Belle
m11: The lost world : Jurassic Park Genres: Action, Adventure, Sci-Fi; Rating: 6.5; Year: 1997
p11: Naomie Harris
d2: Roland Emmerich
p12: Mads Mikkelsen
m12: Golden Eyes Genres: Action, Adventure, Thriller; Rating: 7.2; Year: 1995
p13: Judi Dench
m16: Skyfall Genres: Action, Adventure, Thriller Rating: 7.8; Year: 2012
p15: Thomas F. Duffy
p14: Pierce Brosnan
e1
m17: Independence Day Genres: Action, Adventure, Sci-Fi; Rating: 6.9; Year: 1996
m13: Tomorrow Never Dies Genres: Action, Adventure, Thriller; Rating: 6.5; Year: 1997
m14: The World Is Not Enough Genres: Action, Adventure, Crime, Thriller; Rating: 6.4; Year: 1999
m15: Casino Royale Genres: Action, Adventure, Thriller; Rating: 8.0; Year: 2006
P1
P2
M1 rating>7.0; action, t1
M2 rating>7.0; adventure, t2
M* rating>7.0; action & adventure t>t1, t>t2
P
M1 rating>7.0; action, t1
M rating>7.0; action & adventure t>t1
V1
P
M1 rating>7.0; adventure, t2
M rating>7.0; action & adventure t>t2
V2

(b) Pattern query Q (c) Views V (a) Snapshot of collaboration network G of IMDb

Figure 1: Pattern query Q, Views $\mathcal{V}$ and collaboration network G

Suppose that one is looking for movies that he is interested in, then the search conditions can be expressed as a pattern query Q (Fig. 1(b)) as follows: (1) movies M should have high ratings, *e.g.,* $r > 7.0$, and are with genres "action" and "adventure"; (2) the M should be played by experienced performers P_1 and P_2. Specifically, P_1 (resp. P_2) played movie M_1 (resp. M_2) with $r > 7.0$, $g =$ "action" (resp. $g =$ "adventure") and $t_1 < t$ (resp. $t_2 < t$), where t (resp. t_1, t_2) is the release time of the M (resp. M_1, M_2).; and (3) the M is marked as "output node" with "*", *i.e.,* users only require the matches of M to be returned as search results.

The matches of Q, denoted as $M(Q, G)$, consists of a set of subgraphs in G that are isomorphic to Q. For example, $M(Q, G)$= $\{\{(\text{P}_1,\text{p}_{11})(\text{P}_2,\text{p}_2)(\text{M}_1,\text{m}_9)(\text{M}_2,\text{m}_i)(\text{M},\text{m}_8)|i \in [3, 5]\}$, $\{(\text{P}_1,\text{p}_{13})(\text{P}_2,\text{p}_{11})(\text{M}_1,\text{m}_{12})(\text{M}_2,\text{m}_j)(\text{M},\text{m}_{16})|j \in [8, 9]\}$, $\{(\text{P}_1,\text{p}_{11})(\text{P}_2,\text{p}_{13})(\text{M}_1,\text{m}_k)(\text{M}_2,\text{m}_{12})(\text{M},\text{m}_{16})|k \in [8, 9]\}\}$. Observe that (1) it takes $O(|G|!|G|)$ time to compute $M(Q, G)$, where $|G|$ is the size of G [3]; due to high computational cost, optimization techniques, *e.g.,* view based evaluation, are needed to speed up query evaluation; (2) since the graph G is distributively stored, no match can be found in a single site, which indicates that data has to be shipped from one site to another to find matches. With this comes the need for distributive techniques for graph pattern matching; (3) as the "query focus" of Q is M, "At World's End" and "Skyfall" are returned as query results. While in practice, users may be interested in the best matches, rather than the whole set of matches of "query focus" M, then a metric is needed to rank matches. For example, compared with "At World's End", "Skyfall" and its corresponding isomorphic subgraph have higher comprehensive rating,

Series ISSN: 2367-2005

10.5441/002/edbt.2017.65

which makes it a better match than “At World’s End”. □

In light of these, we present MovieFinder, a novel system to effectively identify movies in social networks via top-k graph pattern matching. In contrast to previous graph search systems (see [7] for a survey), MovieFinder (1) supports graph pattern matching with subgraph isomorphism [3], and combines graph pattern matching with result ranking, (2) evaluates top-k graph pattern matching in a parallel manner, and (3) optimizes local evaluation by using materialized views, and maintains views via incremental techniques [6].

To the best of our knowledge, MovieFinder is among the first efforts to search movies on large and distributed social networks via graph pattern matching. It should also be remarked that movie searching is just one application of the technique, one may apply the technique to find *e.g.,* people, hotels, restaurants and so on.

2. DISTRIBUTED TOP-K GRAPH PATTERN MATCHING

We first review the notion of subgraph isomorphism. We then introduce graph fragmentation, followed by the problem of distributed top-k graph pattern matching.

Subgraph isomorphism. Given a data graph $G = (V, E, f_A)$ and a pattern query $Q = (V_p, E_p, f_v)$, a match of Q in G via subgraph isomorphism is a subgraph G_s of G that is isomorphic to Q, *i.e.,* there is a bijective function h from V_p to the node set of G_s such that (1) for each node $u \in V_p$, $f_v(u) = f_A(h(u))$; (2) (u, u') is an edge in Q if and only if $(h(u), h(u'))$ is an edge in G_s. We denote by $G[M(Q, G)]$ to be the union of all the matches G_s in $M(Q, G)$.

To find matches of query focus, we extend Q by specifying one node in Q as output node, denoted as u_o. Then, the answer to Q in G, denoted by $M(Q, G, u_o)$, is the set of nodes $h(u_o)$, that match the output node u_o of Q in G_s, for all matches G_s of Q in G.

Distributed graphs. A fragmentation $\mathcal{F}$ of a graph $G = (V, E, f_A)$ is $(F_1, \cdots, F_n)$, where each fragment F_i is specified by $(V_i \cup F_i.O, E_i, f_{A_i})$ such that (1) $(V_1, \cdots, V_n)$ is a partition of V; (2) $F_i.O$ is the set of nodes v' such that there exists an edge $e = (v, v')$ in E, $v \in V_i$ and node v' is in another fragment; we refer to v' as a virtual node and e as a crossing edge; and (3) $(V_i \cup F_i.O, E_i, f_{A_i})$ is a subgraph of G induced by $V_i \cup F_i.O$. We assume *w.l.o.g.* that each F_i is stored at site S_i for $i \in [1, n]$.

Distributed Top-k Graph Pattern Matching. Given an integer k, a pattern query Q with output node u_o and a fragmentation $\mathcal{F}$ of a graph G, the distributed top-k graph pattern matching problem is to find the best k matches to u_o of Q in G.

We next show how MovieFinder supports distributed top-k graph pattern matching via parallel computation that integrates asynchronous message passing with optimized local evaluation.

3. THE SYSTEM OVERVIEW

The architecture of the MovieFinder, shown in Fig. 2, consists of the following three components. (1) A *Graphical User Interface* (GUI), which provides a graphical interface to help users formulate pattern queries, manage data graphs and understand visualized results. (2) A *coordinator* that communicates with GUI and *workers* (to be introduced shortly). Specifically, the *coordinator* (a) forwards various requests, received from GUI, to *workers* for their local processing; (b) assembles partial results from *workers*; (c) ranks matches and returns best k ones as search results. (3) Multiple worker machines (*a.k.a.workers* [4, 8]), which employ Query Executor (QE) to compute local matches, and Incremental Computation Module (ICM) to keep materialized views up to date. We next present the components of MovieFinder and their interactions.

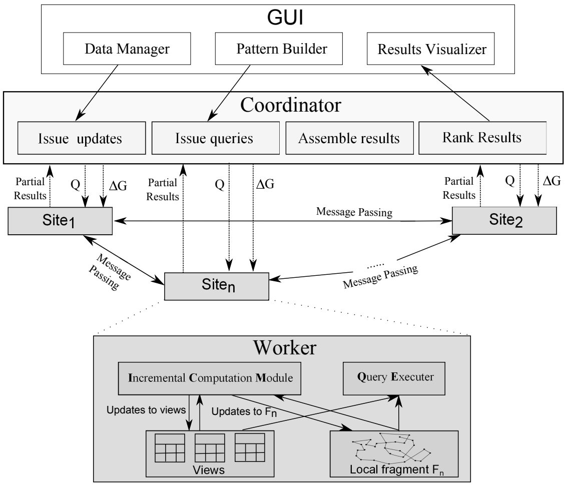


Figure 2: Architecture of MovieFinder

Graphical User Interface. The GUI helps to interact with users, *e.g.,* graph data manipulation, pattern query formulation, and result browse. Specifically, (1) It provides a task-oriented panel to facilitate users to manage graph data. (2) It is equipped with a query panel, which allows users to (a) manually construct a pattern query Q from scratch by drawing a set of query nodes and edges; (b) specify the search conditions of query nodes (*e.g.,* title=“Skyfall”, g=“action & adventure”; $r \leq 7.0$; $t > t_1$); (c) indicate the particular “output” node for which users want to find matches (*e.g.,*M in Example. 1); (d) specify the number k of matches to the “output” node; and (e) designate query target from a list of data graphs. (3) The GUI visualizes query results by layout algorithm, hence the users can browse the matches with more intuition.

Coordinator. The *coordinator* interacts with GUI and *workers* as following. It (1) sends users’ requests, received from GUI to *workers* for their local precessing, and returns query results to GUI for visualization; (2) collects partial results from *workers*, ranks matches based on the ranking metric, and identify best k matches.

Results Ranking. As there may exist a large set of matches of the output node u_o, and users may be only interested in the best k ones. The *coordinator* hence uses a ranking function to identify top-k matches. Intuitively, the ranking function follows one observation from social networks, that’s the higher the rating of v and the total rating of G_s are, the better v is. To be more specific, given a pattern query Q with output node u_o, and a match G_s of Q with node v as the match of u_o, the rank of v is defined as:

$$f(v, u_o) = v.r * \Sigma_{v_i \in G_s} v_i.r$$

where $v.r$ (resp. $v_i.r$) indicates the rating of v (resp. v_i).

Example 2: Recall Example 1, the highest rating of the match in $M(Q, G)$ that contains m_8 (resp. m_{16}) is 8.9+7.3+7.1=23.3 (resp. 7.2+7.3+7.8=22.3). Then “Skyfall” makes the top-1 match since $f(\mathsf{m}_{16}, u_o) = 7.8 * 22.3 = 173.94$ is greater than $f(\mathsf{m}_8, \mathsf{u}_o) = 7.1 * 23.3 = 165.43$. □

Note that, though we used node attribute, *e.g.,*, movie rating, to define $f()$, while in general cases, other metrics which can be used to measure the “goodness” of matches can also be applied, and readily supported by the system.

Workers. Each *worker* has two modules: Query Executor (QE) and Incremental Computation Module (ICM).

Query Executor. The main task of the QE is query evaluation. As local information may not be sufficient to find matches, and query evaluation is computational expensive, the QE hence (1) applies multithreaded computation to collect necessary information from other sites, and integrates collected information with current fragment to conduct local evaluation; and (2) employs view-based technique to optimize evaluation of graph pattern matching.

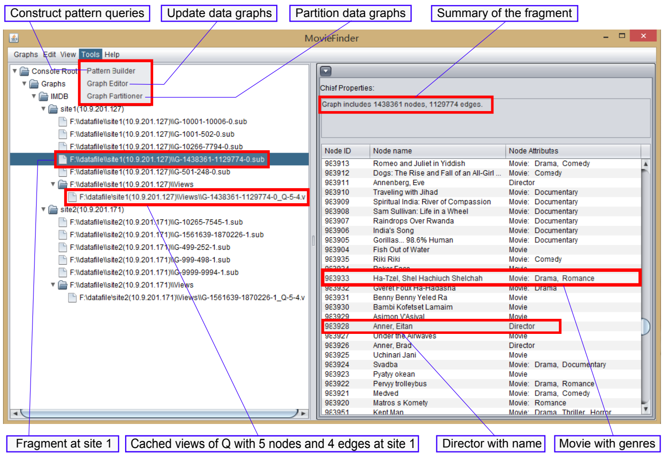
(1) Local evaluation. Upon receiving pattern query Q from *coordinator*, the QE starts one thread to do the following. (a) It checks whether each virtual node v at current fragment F_i is a candidate match of some pattern node u, *i.e.,* v satisfies search conditions specified by u. (b) For each candidate match v, it then sends node pair $\langle u, v\rangle$ to the site S_j, where v accommodates; and requests the subgraph $G^N_j(u, v)$ of fragment F_j, where $G^N_j(u, v)$ contains neighborhood information of v in F_j, (see below for more details about computation of $G^N_j(u, v)$). (c) After all the $G^N_j(u, v)$ are received and merged with F_i, the QE computes matches with algorithm VF2 [3], and sends local results to the coordinator.

To response requests from other sites such that local evaluation can be processed in parallel at each site, the QE at site S_j constantly waits for messages from other sites, and initializes new threads to compute $G^N_j(u, v)$ when receiving messages $\langle u, v\rangle$ from other sites. Specifically, when message $\langle u, v\rangle$ sent from other site is received by site S_j, a new thread is started by the QE at S_j to conduct restricted breadth first search from v and u in F_j and Q, respectively. For any node v' (resp. u') encountered during the traversal in F_j (resp. Q), if v' is a candidate match of u', then v' is inserted in $G^N_j(u, v)$, and also connected to its neighbor nodes, which are already in $G^N_j(u, v)$.

Example 3: Recall pattern query Q in Example 1. Upon receiving Q, the QE at S_2 identifies m_{15} and m_{16} as the candidate match of the pattern nodes M_1, M_2 and M, and sends node pairs $\langle \mathsf{M}_1, \mathsf{m}_{15}\rangle$, $\langle \mathsf{M}_2, \mathsf{m}_{15}\rangle$, $\langle \mathsf{M}, \mathsf{m}_{15}\rangle$, $\langle \mathsf{M}_1, \mathsf{m}_{16}\rangle$, $\langle \mathsf{M}_2, \mathsf{m}_{16}\rangle$, $\langle \mathsf{M}, \mathsf{m}_{16}\rangle$ to S_3. Once receiving requests, S_3 computes $G^N_3(u, v)$ as response, *e.g.,* $G^N_3(\mathsf{M}, \mathsf{m}_{16})$, which includes two edges $(\mathsf{p}_{13}, \mathsf{m}_{12})$ and $(\mathsf{p}_{13}, \mathsf{m}_{16})$ are returned to S_2. After receiving the response, the QE at S_2 then merges $G^N_3(u, v)$ with F_2, invokes VF2 to compute $M(Q, F_2)$, and sends result $\{(\mathsf{P}_1, \mathsf{p}_{11})(\mathsf{P}_2, \mathsf{p}_{13})(\mathsf{M}_1, \mathsf{m}_i)(\mathsf{M}_2, \mathsf{m}_{12})(\mathsf{M}, \mathsf{m}_{16}) | i \in [8, 9]\}$ to the coordinator. □

(2) Optimization technique. As local evaluation involves subgraph isomorphism checking, which is an NP-complete problem and often computationally expensive, MovieFinder caches query results of commonly issued pattern queries at *workers* and adopts view-based technique to optimize local evaluation.

Suppose a set of view definitions $\mathcal{V} = \{\mathsf{V}_1, \cdots, \mathsf{V}_n\}$ have their extensions $M(\mathcal{V}, F_i) = \{M(\mathsf{V}_1, F_i), \cdots, M(\mathsf{V}_n, F_i)\}$ cached at site S_i. Given pattern query Q, the QE at site S_i computes matches of Q using $\mathcal{V}$ and $M(\mathcal{V}, F_i)$ as following. It first verifies whether Q can be answered using $\mathcal{V}$ by checking whether Q is the same as the union of $Q[M(\mathsf{V}_k, Q)]$ ($k \in [1, n]$). If Q can be answered by using $\mathcal{V}$, the algorithm Match, which takes Q, $\mathcal{V}$ and $M(\mathcal{V}, F_i)$ as input is then invoked to compute matches. Specifically, Match first initializes an empty pattern query $\mathsf{Q_s}$ and an empty set S as the match set of $\mathsf{Q_s}$. It then iteratively invokes Procedure Merge to "merge" $\mathsf{Q_s}$ with V_k, and matches in S with matches in $M(\mathsf{V}_k, F_i)$. In particular, Merge checks whether matches m_1 of $\mathsf{Q_s}$ can be merged with matches m_2 of V_k following the mapping λ that guides the "merge" of $\mathsf{Q_s}$ and V_k. If so, a new match m_0 of the newly formed pattern query $\mathsf{Q_s}$ (merged with V_k) is formed by merging m_1 with m_2, and the set S is updated by replacing m_1 with m_0. When the termination condition, *i.e.,* $\mathsf{Q_s} = Q$ is met, the set S is returned as the match set of Q at S_i.



```
Console Root
  Graphs
    IMDB
      site1(10.9.201.127)
        F:\\datafile\\site1(10.9.201.127)\\\G-10001-10006-0.sub
        F:\\datafile\\site1(10.9.201.127)\\\G-1001-502-0.sub
        F:\\datafile\\site1(10.9.201.127)\\\G-10266-7794-0.sub
        F:\\datafile\\site1(10.9.201.127)\\\G-1438361-1129774-0.sub
        F:\\datafile\\site1(10.9.201.127)\\\G-501-248-0.sub
        F:\\datafile\\site1(10.9.201.127)\\\Views
          F:\datafile\site1(10.9.201.127)\\Views\\G-1438361-1129774-0_Q-5-4.v
      site2(10.9.201.171)
        F:\\datafile\\site2(10.9.201.171)\\\G-10265-7545-1.sub
        F:\\datafile\\site2(10.9.201.171)\\\G-1561639-1870226-1.sub
        F:\\datafile\\site2(10.9.201.171)\\\G-499-252-1.sub
        F:\\datafile\\site2(10.9.201.171)\\\G-999-498-1.sub
        F:\\datafile\\site2(10.9.201.171)\\\G-9999-9994-1.sub
        F:\\datafile\\site2(10.9.201.171)\\\Views
          F:\datafile\site2(10.9.201.171)\\Views\\G-1561639-1870226-1_Q-5-4.v
```

Chief Properties:

Graph includes 1438361 nodes, 1129774 edges.

Node ID	Node name	Node Attributes
983913	Romeo and Juliet in Yiddish	Movie: Drama, Comedy
983912	Dogs: The Rise and Fall of an All-Girl ...	Movie: Comedy
983911	Annenberg, Eve	Director
983910	Traveling with Jihad	Movie: Documentary
983909	Spiritual India: River of Compassion	Movie: Documentary
983908	Sam Sullivan: Life in a Wheel	Movie: Documentary
983907	Raindrops Over Rwanda	Movie: Documentary
983906	India's Song	Movie: Documentary
983905	Gorillas... 98.6% Human	Movie: Documentary
983904	Fish Out of Water	Movie
983935	Riki Riki	Movie: Comedy
983934	Poker Face	Movie
983933	Ha-Tzel, Shel Hachiuch Shelchah	Movie: Drama, Romance
983932	Gveret Foux Ha-Hadasha	Movie: Drama
983931	Benny Benny Yeled Ra	Movie
983930	Bambi Kofetset Lamaim	Movie
983929	Asimon V'Asival	Movie
983928	Anner, Eitan	Director
983927	Under the Airwaves	Movie
983926	Anner, Brad	Director
983925	Uchinari Jani	Movie
983924	Svadba	Movie: Drama, Documentary
983923	Pyatyy okean	Movie
983922	Pervyy trolleybus	Movie: Drama, Romance
983921	Medved	Movie: Drama, Comedy
983920	Matros s Komety	Movie: Romance
983951	Kent Man	Movie: Drama, Thriller, Horror

Figure 3: Visual interface: MovieFinder Manager

Example 4: Recall view definitions $\mathcal{V} = \{\mathsf{V}_1, \mathsf{V}_2\}$, shown in Fig. 1(c), their extensions $M(\mathcal{V}, F_3)$ at S_3 are listed in table below.

View definitions	Extensions
V_1	$\{(\mathsf{P}_1, \mathsf{p}_{13})(\mathsf{M}_1, \mathsf{m}_{12})(\mathsf{M}, \mathsf{m}_{16})\}$
V_2	$\{(\mathsf{P}_1, \mathsf{p}_{13})(\mathsf{M}_1, \mathsf{m}_{12})(\mathsf{M}, \mathsf{m}_{16})\}$

At site S_3, the QE computes matches of Q (see Fig. 1(b)) using $\mathcal{V}$ and $M(\mathcal{V}, F_3)$, as following. (1) It first determines that Q can be answered using $\mathcal{V}$ since Q is the same as $\bigcup_{i\in[1,2]} Q[M(\mathsf{V}_i, Q)]$. (2) It then invokes Match to compute matches. Since no match of V_1 and V_2 can be merged, following the mapping which guides the merge of V_1 and V_2, then no match of Q exists at site S_3. □

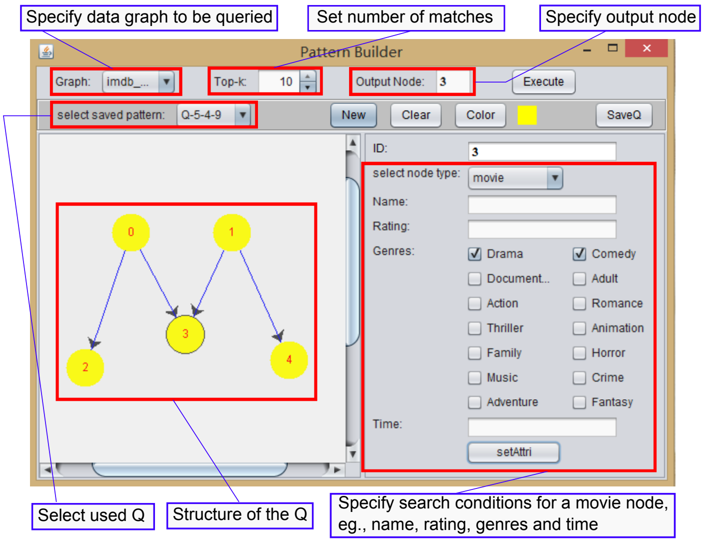
Incremental Computation Module. Real-life social networks change constantly, hence the cached views $M(\mathcal{V}, F_i)$ at site S_i need to be updated, in response to the changes to F_i. However, due to that subgraph isomorphism is computationally expensive and the input, *i.e.,* F_i, is often large, it is costly to recompute $M(\mathsf{V}, F_i \oplus \Delta F_i)$ for each $\mathsf{V} \in \mathcal{V}$, where $F_i \oplus \Delta F_i$ denotes F_i updated by ΔF_i. Instead of recomputation, the ICM incrementally identifies changes to $M(\mathcal{V}, F_i)$, in response to ΔF_i. As ΔF_i is often small in practice, the incremental computation hence is far more efficient than batch computation. The ICM applies the incremental subgraph isomorphism algorithm of [6] to update cached views, for both unit and batch updates.

Example 5: Recall Q, G in Example 1. Suppose that an edge e_1 (marked in red in Fig 1(a)) is inserted into G , then the change to G incurs four new matches: $\{(\mathsf{P}_1, \mathsf{p}_{13})(\mathsf{P}_2, \mathsf{p}_{11})(\mathsf{M}_1, \mathsf{m}_{15})(\mathsf{M}_2, \mathsf{m}_i)(\mathsf{M}, \mathsf{m}_{16}) | i \in [8, 9]\}$, and $\{(\mathsf{P}_1, \mathsf{p}_{11})(\mathsf{P}_2, \mathsf{p}_{13})(\mathsf{M}_1, \mathsf{m}_j)(\mathsf{M}_2, \mathsf{m}_{15})(\mathsf{M}, \mathsf{m}_{16}) | j \in [8, 9]\}$. Instead of recomputing $M(Q, G \oplus \Delta G)$ from scratch, the ICM only visits nodes that are 3 hops away from p_{13}, and identifies the new matches. □

Remark. The MovieFinder identifies all the matches of Q by exact algorithms, *i.e.,* VF2 or our view-based technique, at all *workers*, hence can find top-k matches of u_o with 100% accuracy.

4. DEMONSTRATION OVERVIEW

The demonstration is to show the following: (1) the use of GUI to formulate pattern queries and browse query results; (2) the efficiency of computation of $M(Q, G)$ and top-k matches of u_o when G is distributively stored; (3) effectiveness of view-based optimization technique employed by the QE; and (4) efficiency of the incremental technique applied by the ICM.



Specify data graph to be queried

Set number of matches

Specify output node

Select used Q

Structure of the Q

Specify search conditions for a movie node, eg., name, rating, genres and time

Graph: imdb_...

Top-k: 10

Output Node: 3

select saved pattern: Q-5-4-9

ID: 3

select node type: movie

Name:

Rating:

Genres: ☑ Drama ☑ Comedy □ Document... □ Adult □ Action □ Romance □ Thriller □ Animation □ Family □ Horror □ Music □ Crime □ Adventure □ Fantasy

Time:

Figure 4: Visual interface: Pattern Builder

Setup. To show the performance of MovieFinder, we used a fraction of IMDb [2] with $|V|$=1.1M, $|E|$=1.7M, randomly partitioned it into a set of fragments controlled by the number of fragments $|\mathcal{F}|$. The system is implemented in Java and deployed with fragments on a cluster of 8 machines with 2.9GHz CPU, 8GB Memory.

Interacting with the GUI. We invite users to use the GUI, from pattern query construction to intuitive illustration of query results.

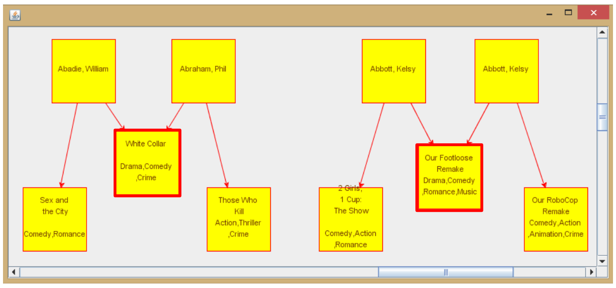
(1) The Manager panel, which is the main control panel of MovieFinder, is used to manipulate the system. As shown in Fig. 3, users can access each module of the MovieFinder as listed in the Tools menu, view both summarized and detailed information, *e.g.,* fragment summary, node attributes, of the selected site.

(2) The Pattern Builder (PB) panel, shown in Fig. 4, facilitates users' construction of pattern queries. Specifically, the PB (a) provides users with a canvas to create new query nodes (resp. edges), (b) allows users to specify search conditions on the query nodes, set output node u_o and the number k of its matches, and (c) supports users to save pattern queries, and reuse them afterwards. For example, a pattern query Q, shown in Fig. 4, is constructed to find movies that are (a) with genres "Drama" and "Comedy", (b) played by people (marked by node "0") who had performed "Romance" movies (marked by node "2"), and (c) directed by people (marked by node "1") who had directed "Action" movies (marked by node "4"). The query focus is marked as "output" node with dark border (node "3"). The pattern query Q can be saved for future use if it is frequently issued.

(3) The GUI provides intuitive ways to help users interpret query results. In particular, the GUI allows users to browse (a) all the matches *w.r.t.* Q, and (b) top-k matches *w.r.t.* u_o. As an example, the query results of Q, given in Fig. 4, are shown in Fig. 5, and the top-2 movies, *i.e.,* "White Collar" and "Our Footloose Remake" are marked with thickened border.

Performance of query evaluation. We also aim to show (a) the performance of the parallel computation supported by the MovieFinder, and (b) the performance of Query Executor (QE) and Incremental Computation Module (ICM) supported by workers.

Performance of parallel computation. We will show efficiency and scalability of parallel computation supported by MovieFinder. As will be seen, when the number $|\mathcal{F}|$ of sites increases from 4 to 8, the query time is reduced by 35%, in average.



Abadie, William
Abraham, Phil
Abbott, Kelsy
Abbott, Kelsy
White Collar Drama,Comedy ,Crime
Our Footloose Remake Drama,Comedy ,Romance,Music
Sex and the City Comedy,Romance
Those Who Kill Action,Thriller ,Crime
2 Girls, 1 Cup: The Show Comedy,Action ,Romance
Our RoboCop Remake Comedy,Action ,Animation,Crime

Figure 5: Visual interface: Query results

Performance of QE. We will show (a) the efficiency of QE by reporting its performance on IMDb; and (b) how substantial the performance is improved when view-based technique is applied. We show that in average the query time can be reduced by 70% with optimization technique.

Performance of ICM. We will also show the improvement of the ICM compared to batch computation that recomputes the materialized views in response to updates. In particular, we will report the performance of incremental computation by varying data graphs with unit update (single edge insertion/deletion) as well as batch updates (a list of edge insertions/deletions). As will be seen, the ICM performs significantly better than its batch counterparts, when data graphs are changed up to 30%.

Summary. This demonstration aims to show the key ideas and performance of the movie search system MovieFinder, based on the technique of distributed top-k graph pattern matching. The MovieFinder is able to (1) evaluate pattern queries defined in terms of subgraph isomorphism in parallel and identify top-k movies on large, distributively stored social networks; (2) efficiently compute matches with view-based technique; (3) incrementally maintain materialized views for dynamic social graphs; and (4) facilite users' use and understaing with intuitive graphical interface. These together convince us that the MovieFinder can serve as a promising tool for movie search on real-life social networks.

Acknowledgments. Wang is supported in part by NSFC 61402383 and 71490722, Sichuan Provincial Science and Technology Project 2014JY0207, and Fundamental Research Funds for the Central Universities, China.

5. REFERENCES

[1] Facebook statistics; visited december 2016. *http://expandedramblings.com/index.php/by-the-numbers-17-amazing-facebook-stats/.*

[2] Imdb dataset. *http://www.imdb.com/interfaces.*

[3] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *TPAMI*, 26(10):1367–1372, 2004.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150, 2004.

[5] W. Fan. Graph pattern matching revised for social network analysis. In *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*, pages 8–21, 2012.

[6] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. Incremental graph pattern matching. In *SIGMOD*, 2011.

[7] T. Lappas, K. Liu, and E. Terzi. A survey of algorithms and systems for expert location in social networks. In *Social Network Data Analytics*. 2011.

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

VAT: A System for Data-Driven Biodiversity Research

Christian Beilschmidt Johannes Drönner Michael Mattig Bernhard Seeger
Dept. of Mathematics and Computer Science
University of Marburg, Germany
{beilschmidt, droenner, mattig, seeger}@mathematik.uni-marburg.de

ABSTRACT

Visual analytics plays a leading role in data-driven research. This requires systems for fast and intuitive data exploration. In this paper we demonstrate VAT, a system for Visualizing, Analyzing and Transforming spatio-temporal data. The system consists of a distributed back end for low-latency processing and a web front end that allows creating workflows of computations in an exploratory fashion. A novel quality of the system is the combination of scientific processing while simultaneously tracking the provenance of the data and aggregating a list of data citations. These features make a visual analytics approach for large, heterogeneous spatio-temporal data feasible.

CCS Concepts

•**Information systems → Geographic information systems;** *Data analytics; Information integration;* •**Human-centered computing → Visualization;**

Keywords

Scientific Workflows, Provenance, Interactive Analysis

1. INTRODUCTION

Visual analytics plays a leading role in data-driven research. Especially in geoscience, researchers investigate spatio-temporal data by means of interactive exploration. As data sizes increase rapidly, there is a growing demand for exploratory tools with fast response time. To gain insights from the data, researchers want to examine different research ideas by expressing queries and evaluating the delivered results. However, there are multiple challenges for a scientist as detailed in the following.

Data from the geoscience domain is inherently heterogeneous. There are several data formats for vector and raster data as well as different reference systems for space and time. This makes data integration and data correlation a very cumbersome and time-consuming tasks for researchers even before tackling the actual research problem.

Scientific work itself and also the recent trend of journals to encourage data sharing make correct citations of data sources indispensable. Furthermore, it is necessary to ensure validity and reproducibility of computations. Both tasks become hard to accomplish when working in an exploratory fashion. As new ideas for additional data processing steps arise mostly when reviewing intermediate results, an upfront specification of the whole computation is not feasible. Recreating the computation steps afterwards is laborious and error-prone. To solve this, a system that offers scientific data processing should keep track of the whole path of processing steps also known as workflows. In addition, all references of incorporated source data should be aggregated as a list of citations.

To cope with these challenges, we demonstrate the VAT system in this paper, a system for Visualizing, Analyzing and Transforming spatio-temporal data in biodiversity science. It facilitates interactive data exploration and cleansing by creating and executing so-called exploratory workflows. For this, it offers processing building blocks for filtering, transforming, visualizing, and creating statistics. It enables users to join heterogeneous data and to work with time-series. They can (1) visualize data, (2) export data in consolidated formats for further analysis in custom tools, and (3) share reproducible workflows.

The VAT system is already in use in GFBio, a German national infrastructure project for managing, archiving, and providing access to biodiversity data [5]. The project aims to provide a sustainable service architecture for German research projects in biodiversity science. VAT enables researchers to identify interesting scientific topics, geographic regions and time spans by providing added value services for data visualization and analysis. It furthermore aims to facilitate reproducibility and data re-usage.

In the following, Section 2 gives an overview of the VAT system's architecture. Section 3 summarizes the functionality of the system and Section 4 describes the proposed demonstration scenario. Section 5 concludes the paper.

2. ARCHITECTURE

VAT has a client-server architecture that consists of a back end called MAPPING and a web-based front end WAVE. (c.f. Figure 1). The complete architecture is described in more detail in our previous work [1, 2].

MAPPING (Marburg's Analysis, Processing, and Provenance of Information for Networked Geographics) is a dis-

Series ISSN: 2367-2005

10.5441/002/edbt.2017.66

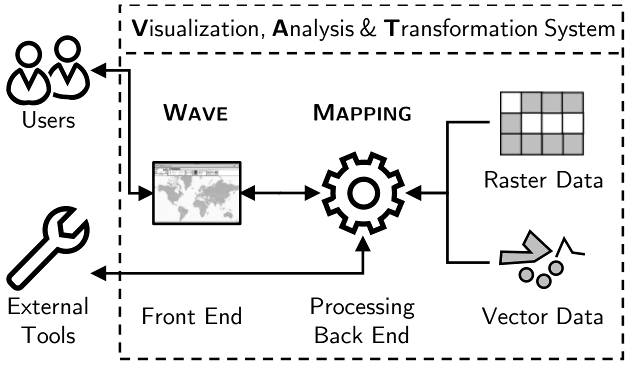


Figure 1: A condensed view on the system architecture

tributed scientific workflow processing system for low-latency processing of spatio-temporal data. It includes a workflow processing engine and various operators written in C++. For the sake of performance improvement, MAPPING utilizes OpenCL to massively parallelize parts of the processing on the GPU. It also manages heterogeneous data and allows processing of raster as well as vector data. For easy access, MAPPING implements important parts of the standardized OGC[1] protocols. This allows many tools to access computation results via a standard interface.

WAVE (Workflow, Analysis and Visualization Editor) is an interactive web application for visual analytics and data cleansing which creates exploratory workflows [3]. It offers a reactive user interface where users apply actions on data via operators and review the results. The interface builds up on Angular 2[2] and OpenLayers 3[3], and uses an implementation of Google's Material Design components to offer appealing and touch-compatible control elements for desktop and mobile usage.

3. FUNCTIONALITY

The main scope of VAT is to support visualizing and processing collections of geo objects. These are points, lines, polygons and rasters. Examples for these data types are species occurrences for points, rivers for lines, forest regions for polygons and temperature grids for rasters. Additionally, each object has a temporal validity.

3.1 Exploring Data

WAVE provides data visualization and interactive data exploration by applying operators. Operators fall into three categories: There are source operators that allow including data either from a repository of hosted environmental raster data and species related vector data, or from custom CSV files. Then, there are operators for filtering, combining and transforming data, e.g. attribute or point-in-polygon filters. Finally, there are statistics operators that allow creating figures like scatter plots and histograms. Additionally, the user can incorporate R scripts to extend the statistics functionality. The first two operator categories produce object collections that are presented in the form of layers on a map and a data table (arranged above each other, c.f. Figure 2). WAVE presents the results of the latter as plots on a sidebar of the application.

For displaying large object collections, VAT offers data reduction techniques by compressing raster and vector data. It computes raster images in preview resolutions to reduce response times. As there is only a limited amount of pixels available on the user's screen, the loss of accuracy has no impact on the visualization. VAT uses a visual clustering approach (an adaptation of the method presented by Jänicke, et al. [6]) to reduce the amount of point data to be transferred and visualized. This technique facilitates recognizing the density of the data objects on the map. The data table shows aggregates of theses clusters for non-spatial attributes. Both techniques provide more accurate results when processing data of smaller areas. This means, zooming into interesting data reveals more detailed information and is therefore the intended exploration method. In the end, for scientificly valid results, VAT offers the functionality to compute the whole workflow in full resolution.

3.2 Combining Heterogeneous Data

In VAT, each object of a collection has three components: a spatial reference, a temporal reference and attributes. The spatial reference specifies a geometric object (e.g. a point) that corresponds to a coordinate reference system. The temporal reference specifies an interval from start to end time using a reference system (e.g. the Gregorian Calendar). The list of attributes contains different data types (e.g. strings or floating points). The spatial and temporal reference system is uniform for all objects within a collection. Because of the presence of temporal references in each object collection, a collection is considered as a time series. In a raster time series, each grid of cells has the same spatial and temporal reference.

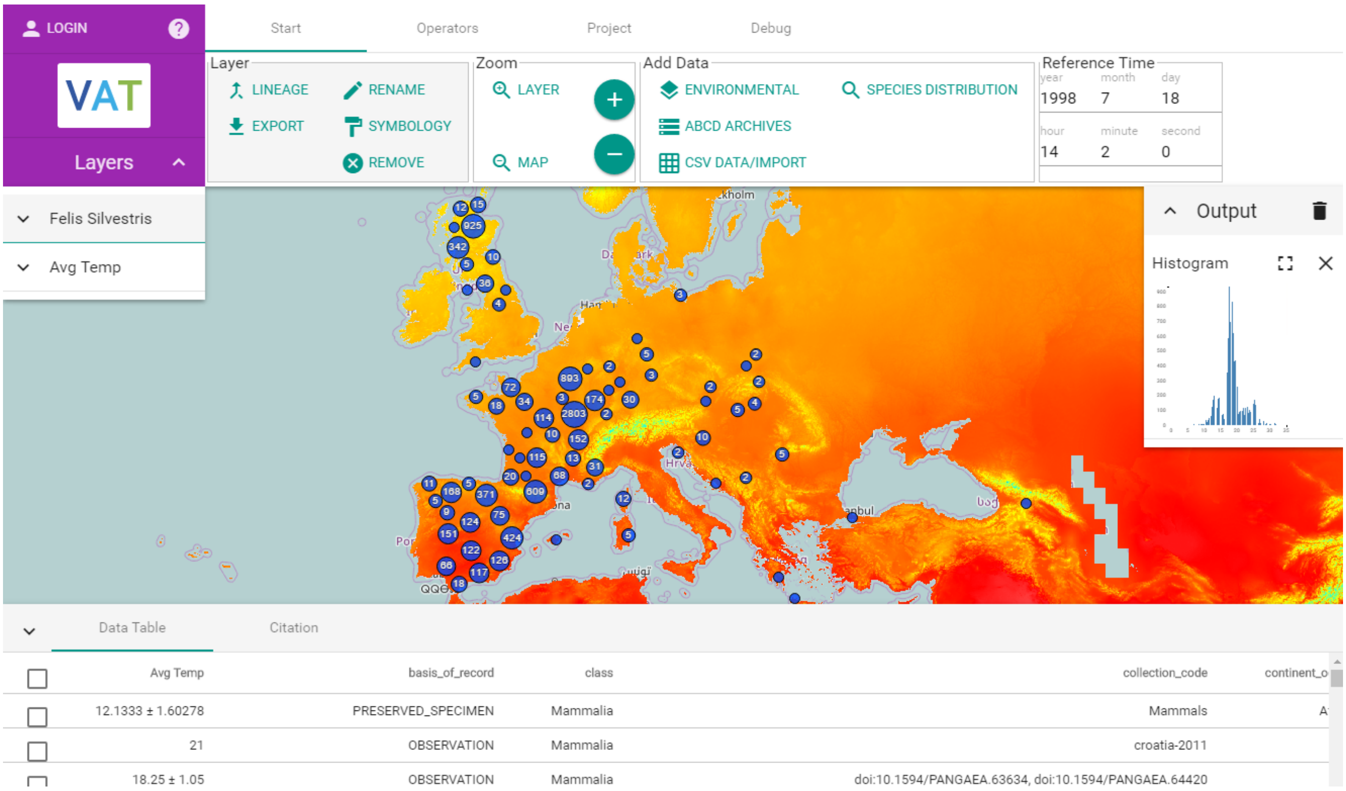
To join object collections, the data needs to be in a unified reference systems. MAPPING offers operators to transform data of one reference system into another. While this is usually very cumbersome for the user, WAVE automatically applies these transformations whenever necessary. This makes it easy to apply operators to join initially heterogeneous object collections. Additionally, WAVE suggests and restricts valid operator inputs (e.g. users can only select points and polygons for a point-in-polygon check).

Every combination operator has to consider the time series semantics. An example is the combination of a raster time series of monthly temperatures with point data (which have irregular time intervals). An input point has to be split into multiple points with different time intervals, if and only if it overlaps at least the end of one month. For instance, if an input point is valid from the first of January to the end of February, it will result in one point with temperatures from January and one from February.

Another example is to compare the temperature of the current date with the temperature of the same day of the previous year. For this, MAPPING offers temporal operators for shifting the temporal context. By shifting relatively one year to the past, VAT can compute a difference expression on a single raster time series.

WAVE uses a user-defined point in time to visualize the data of a time series. It uses it to select a time-slice of the series and retrieve only the data objects with matching validity. When a user changes the time, WAVE triggers an update for each view, i.e. the map, the data table and the

[1] Open Geospatial Consortium, www.opengeospatial.org
[2] www.angular.io
[3] www.openlayers.org



Reference Time

year	month	day
1998	7	18

hour	minute	second
14	2	0

	Avg Temp	basis_of_record	class	collection_code	continent_o
□	12.1333 ± 1.60278	PRESERVED_SPECIMEN	Mammalia	Mammals	A
□	21	OBSERVATION	Mammalia	croatia-2011	
□	18.25 ± 1.05	OBSERVATION	Mammalia	doi:10.1594/PANGAEA.63634, doi:10.1594/PANGAEA.64420	

Figure 2: An overview of WAVE. The central map component shows clustered point data and a raster. Below is the data table. On top is a menu bar for data and operator selection. On the left-hand side is a list of map layers. On the right-hand side is a plot area containing a histogram.

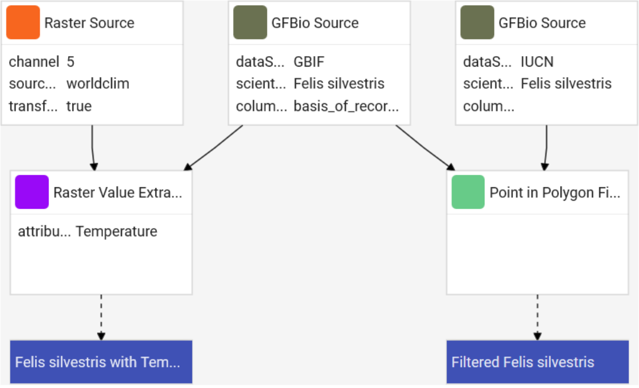


Figure 3: A lineage graph for a small workflow

plots. A video mode for uniform time steps of an interval (e.g. monthly) allows visualizing the changes over time.

3.3 Provenance Tracking

Provenance tracking is of utmost importance for the reproducibility of computations. WAVE allows data exploration by interactively applying operators on data. Users can utilize different views to evaluate results of computations. They can form new ideas and discard dead ends. We use the notion of exploratory workflows that describe the path of computations from the sources to a final result as a rooted tree. WAVE automatically updates corresponding workflows on every user action. The user is able to look up the full processing path at any time in the so-called lineage graph. Figure 3 shows an example of a series of applied operations in an exploratory workflow. The data flows from the source operators at the top to the resulting layers (blue boxes). VAT uses a workflow representation in the human-readable and interchangeable JSON data format.

The workflow data structure makes it furthermore possible to share computations and results with other researchers. This means either publishing fixed parametrized workflow results for reproducibility reasons or configurable workflows that allow other researchers to use validated workflows for their own data processing.

3.4 Collecting Citations

Correctly citing all sources of a workflow is indispensable for scientific work. While this task is complex in generic scenarios when querying database systems [4], VAT can take advantage of the custom implementation of each operator. In VAT, the tracking of citations is an inherent part of every operator's implementation.

More precisely, there is a default method that applies a duplicate eliminating union operator to the citations of the input operators. Certain operators, for instance source operators with filtering option, can change this behavior to only include citations for selected data. However, it is essential to never remove any citation of a data object that is incorporated in generating a result. An example could be a point data subtraction, where a data object is responsible for removing another object and needs therefore to be included in the citation list. MAPPING's workflow framework guarantees this restriction.

3.5 Data Export

When exporting data sets for further usage, VAT bundles a ZIP file containing three components. The first is the result of the workflow in a raster (GeoTIFF) or vector format (CSV or GeoJSON), computed in full resolution. The second is the workflow description itself, containing the parametrization of each operator. The third one is a complete list of aggregated citations. VAT allows several metadata formats for workflow descriptions and citations like CSV or JSON.

4. DEMONSTRATION SCENARIO

In this section, we present two real-world scenarios that exemplify working with exploratory workflows in VAT. Because of the brevity of this paper we will show only the success case. Of course, one can easily imagine that the user took many wrong turns in order to achieve the result. Because of the automatic tracking of the workflow, the user can always trace back the steps.

4.1 Data Cleansing

The user is interested in the distribution of animals of the cat family, e.g. Felis silvestris (wildcat) from GBIF[4] in Europe. For this, the user adds occurrence data from the repository with the intend to cleanse it. The data occurs as a layer on the map and the user recognizes possible outliers by visually inspecting the clustering on the map.

For a first outlier removal (e.g. zoo animals), the user looks up so called expert ranges from IUCN[5] that outline the expected habitat of a species. The user filters the occurrence points by applying the point-in-polygon filter operator using the expert ranges. The result is a new layer which contains all occurrences contained by the expert ranges.

When looking at the data table, there are aggregates of default parameters from GBIF. From literature, the user knows that the species lives between sea level and a certain height. However, there is currently no elevation information present. The user adds hosted elevation raster data from WorldClim[6] as a new layer to WAVE. Then, the user attaches the raster data by applying the raster-value-extraction operator on both layers. The result is an enriched layer that serves then as an input for creating a histogram plot. The user applies a numeric range filter to remove all outlier occurrences which are not in the expected range.

The next step is exporting the resulting cleansed data and inspecting the file. It contains the data, the workflow and all citations that were included into the computation.

4.2 Statistical Analysis of Time Series

This part of the demonstration shows the impact of time series computations by observing bird movements. For this, the user starts in a clean project in WAVE and adds a layer of a migratory bird species, e.g. Sterna paradisaea (Arctic tern) occurrence points, to the map. The user first adds hosted environmental data of averaged monthly temperatures from WorldClim to the map. Then, the user inspects patterns where these birds have clusterings in the world and assumes a movement that is correlated with temperatures. As a third step, the user applies the raster-value-extraction operator to enrich the occurrence points with the corresponding temperatures. The user then applies a temporal operator to form a small temporal interval around the bird occurrences. The associated temperature values in the data table change over time when traversing in monthly intervals. To get a better understanding of the temperature attribute distribution, the user plots a histogram and observes a dense peak in the diagram.

To compare the observation, the user adds data of a resident bird, e.g. Columba oenas (Stock dove), to the map. As this is a species is stationary, there should be different results when using the same workflow. The user simply changes the source of the previous workflow and VAT computes new results. The map and the diagram show significantly different distribution patterns.

5. CONCLUSIONS

In this demo paper we presented VAT, a system for visualizing, analyzing and transforming spatio-temporal data while tracking citations and provenance information. We showed a brief system overview and pointed out the most important features. In our usage scenario, we presented two real-world applications of the field of biodiversity that also reveal interesting research opportunities for the database community.

6. ACKNOWLEDGMENTS

This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) under grant no. SE 553/7-2.

7. REFERENCES

[1] C. Authmann, C. Beilschmidt, J. Drönner, M. Mattig, and B. Seeger. Rethinking Spatial Processing in Data-Intensive Science. In *BTW Workshops*, pages 161–170, 2015.

[2] C. Authmann, C. Beilschmidt, J. Drönner, M. Mattig, and B. Seeger. VAT: A System for Visualizing, Analyzing and Transforming Spatial Data in Science. *Datenbank-Spektrum*, 15(3):175–184, 2015.

[3] C. Beilschmidt, J. Drönner, M. Mattig, M. Schmidt, C. Authmann, A. Niamir, T. Hickler, and B. Seeger. Interactive Data Exploration for Geoscience. In *BTW Workshops*, 2017.

[4] P. Buneman, S. Davidson, and J. Frew. Why Data Citation is a Computational Problem. *Communications of the ACM*, 59(9):50–57, 2016.

[5] M. Diepenbroek, F. O. Glöckner, P. Grobe, A. Güntsch, R. Huber, B. König-Ries, I. Kostadinov, J. Nieschulze, B. Seeger, R. Tolksdorf, et al. Towards an Integrated Biodiversity and Ecological Research Data Management and Archiving Platform: The German Federation for the Curation of Biological Data (GFBio). In *GI-Jahrestagung*, pages 1711–1721, 2014.

[6] S. Jänicke, C. Heine, R. Stockmann, and G. Scheuermann. Comparative Visualization of Geospatial-temporal Data. In *GRAPP/IVAPP*, pages 613–625, 2012.

[4] Global Biodiversity Information Facility, www.gbif.org

[5] The International Union for Conservation of Nature, www.iucn.org

[6] Global Climate Data, www.worldclim.org

Demonstration

SDOS: Using Trusted Platform Modules for Secure Cryptographic Deletion in the Swift Object Store

Tim Waizenegger
University of Stuttgart
Institute for Parallel and Distributed Systems
waizentm@ipvs.uni-stuttgart.de

Frank Wagner
University of Stuttgart
Institute for Parallel and Distributed Systems
wagnerfk@ipvs.uni-stuttgart.de

Cataldo Mega
University of Stuttgart
Institute for Parallel and Distributed Systems
megaco@ipvs.uni-stuttgart.de

ABSTRACT

The secure deletion of data is becoming increasingly important to individuals, corporations as well as governments. Recent advances in worldwide laws and regulations now require secure deletion for sensitive data in certain industries. Data leaks in the public and private sector are commonplace today, and they often reveal data which was supposed to be deleted. Secure deletion describes any mechanism that renders stored data unrecoverable, even through forensic means. In the past this was achieved by destroying storage media or overwriting storage sectors. Both of these mechanisms are not well suited to today's multi-tenant cloud storage solutions.

Cryptographic deletion is a suitable candidate for these services, but a research gap still exists in applying cryptographic deletion to large cloud storage services. Cloud providers today rarely offer storage solutions with secure deletion for these reasons. In this Demo, we present a working prototype for a cloud storage service that offers cryptographic deletion with the following two main contributions:

A key-management mechanism that enables cryptographic deletion an on large volume of data, and integration with Trusted Platform Modules (TPM) for securing master keys.

Keywords

secure data deletion, cryptographic deletion, data erasure, records management, retention management, key management, data shredding, trusted platform module, TPM

1. BACKGROUND

Cloud based storage solutions are popular services today especially among consumers. They are used for synchronizing data across devices, for backup and archiving purposes, and for enabling access at any time from anywhere. But the adoption of such storage services still faces many challenges in the government and enterprise sector. The customers, as well as the providers, have a desire to move storage systems, or parts of these systems, to cloud environments in order to reduce cost and improve the service. But security issues often prevent customers from adopting cloud storage services.

The providers often address these issues by offering some type of data encryption. They differ in three aspects: i) where the data encryption happens, ii) who has authority over the encryption keys, and iii) how keys are managed.

In most offerings, the provider has authority over master keys and encryption happens on the provider side [1]. This allows the provider to read the customer's data and enables them to offer more advanced services and up-sell customers in the future. If client side encryption is used and customers have authority over master keys, no provider access is possible and less trust in the provider is required. Client side encryption with customer side key authority enables the use of cloud storage services for especially sensitive data.

We propose a cloud storage systems that employs client-side encryption of content in order to address confidentiality concerns of customers. We further propose a key management that enables cryptographic deletion in order to assure customers' legal and regulatory compliance.

In this demo, we present a cloud storage system with the two main contributions:

1. Transparent data encryption with support for cryptographic deletion.
2. Trusted Platform Module integration that provides secure deletion and confidentiality for master keys.

2. CRYPTOGRAPHIC DELETION

An often overlooked security aspect of cloud storage systems is the secure deletion of data. Secure deletion describes any mechanism that renders deleted data unrecoverable, even through forensic means. Recent advances in worldwide regulation make secure deletion a requirement in many industries like banking and law enforcement [2, 4, 5]. Even industries without explicit regulation have an interest in securely removing deleted data in order to prevent future leaks and exposure [6]. In the past, secure deletion was achieved by destroying storage media or overwriting storage sectors. Both of these mechanisms are not well suited to today's multi-tenant cloud storage solutions. Identifying the physical disks that need to be destroyed, or the blocks that need to be overwritten, becomes difficult to impossible [7]. In this work, we assume an untrusted cloud storage

Series ISSN: 2367-2005

10.5441/002/edbt.2017.67

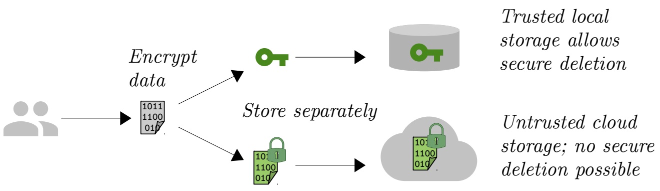

Encrypt data
1011 1100 010
Store separately
Trusted local storage allows secure deletion
Untrusted cloud storage; no secure deletion possible

Figure 1: Cryptographic deletion: Delete data indirectly by securely deleting their encryption key.

system in which all deleted data can be restored later by forensic means. Cryptographic deletion is a suitable candidate for these services, but a research gap still exists in applying cryptographic deletion to large cloud storage services [8]. As shown in Figure 1, cryptographic deletion works by encrypting the data prior to storing it in an untrusted location. The encryption key must be kept in a secure location and can later be deleted in order to indirectly delete the data. This requires that data encryption happens in the customers trusted environment and that the customer has authority over the encryption key. Cryptographic deletion is therefore a function of the key management mechanism in client-side encryption solutions.

2.1 The Key-Cascade

Our "Key-Cascade" key management mechanism is based on a tree structure in which each node contains encryption keys and child nodes are encrypted with a key from their parent [3, 10]. Figure 4 shows an example of this structure with Key 0 as the master key. Each inner node contains a list of encryption keys and each key is used to encrypt one of the node's children. E.g. in Figure 4, Key 1 is used to encrypt the child node containing Keys 17 through 21. Because this child node is encrypted with Key 1, it is called Node 1. The leaves of this tree are nodes containing encryption keys for the actual data objects. Each (encrypted) node is stored as an object inside the object store and identified by its node ID. The IDs for keys, nodes, and objects are used for accessing the Key-Cascade data structure. The IDs are assigned so that only an object ID is needed to calculate the list of node and key IDs along the path to this leaf. This allows decoupling the retrieval and processing of the encrypted nodes.

Figure 2 shows how cryptographic deletion is realized on this data structure. The purpose of the Key-Cascade is to transfer the property of secure deletion from the master key (stored in TPM) to the large number of object keys. This is achieved through the hierarchical dependency between the keys. Once a key becomes inaccessible, all its child nodes become inaccessible as well, leading to cryptographic deletion of the corresponding objects. Once the master key is securely deleted (by the TPM), all the nodes of the tree become inaccessible and all the objects become securely deleted as well.

We use the logarithmic height of the tree in order to cryptographically delete individual objects with minimal overhead: Figure 2 shows how Object 7 is deleted by generating a new (and deleting the old) master key and modifying a path of nodes. The nodes along the path from master key to Object 7 get copied and modified in two ways: i) internal nodes are copied while the key for the child node on this path is replaced by a newly generated one. ii) leaf nodes contain the object keys. This node is copied as well but the object key (which should be deleted) is not copied. In both cases the copied nodes are then re-encrypted with their new parent key. All off-path branches remain unaffected and are now accessible through the new master key and modified nodes.

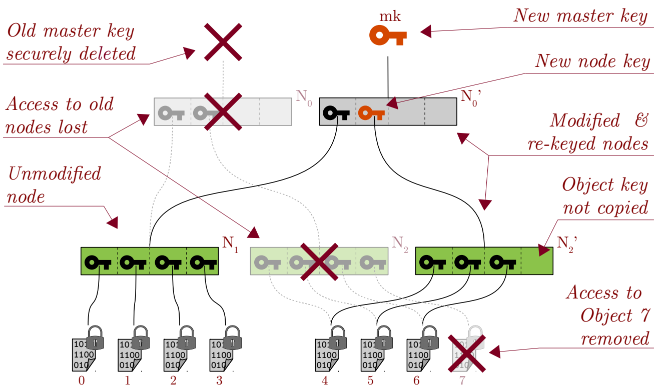


Figure 2: the Key-Cascade data structure after deletion of object 7.

All the old nodes and objects become inaccessible because the old master key was securely deleted by the TPM. It is not possible to restore these deleted objects even if all the old nodes can be restored, because the old master key is no longer available. This recursive "re-keying" of nodes always includes replacing the master key. In order to reduce cost, deletions can be processed in batches. The master key then only needs to be replaced once per batch.

2.1.1 *Key-Cascade properties*

The properties of the Key-Cascade are determined by two parameters: The tree height h and the node size S_n. These properties include the maximum number of object keys, the number of nodes, and the size of the whole data structure. In the following, we give two examples for these properties. Each key has a size of 32 bytes in these calculations because we use the AES256 encryption algorithm.

Example 1:

Tree height $h = 2$, node size $S_n = 2^2 = 4$.

This results in a cascade consisting of 5 nodes with space for 16 object keys. When fully utilized, the Key-Cascade needs 640 bytes to store the nodes. Re-keying requires up to 12 operations. With data objects of 100 kilobytes average size, this cascade can store keys for 1.6 megabytes of data. The cascade therefore imposes a storage overhead of 0.04%

Example 2:

Tree height $h = 3$, node size $S_n = 2^8 = 256$.

This more realistic parameter setting results in a cascade consisting of 65,793 nodes with space for 16,777,216 object keys. When fully utilized, the Key-Cascade needs 514 megabytes to store the nodes. Re-keying requires up to 774 operations (note that these are small in-memory operations). With data objects of 100 kilobytes average size, this cascade can store keys for 1.6 terabytes of data. The cascade therefore imposes a storage overhead of 0.03%

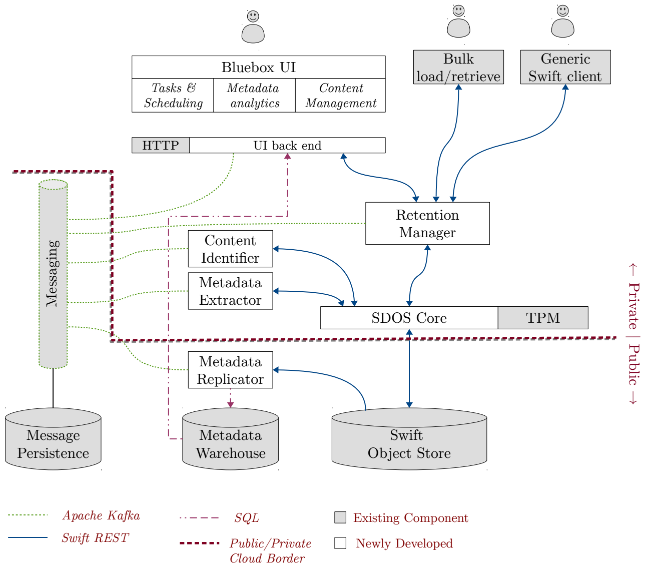


Figure 3: The architecture of the Micro Content Management system with hardware Trusted Platform Module (TPM).

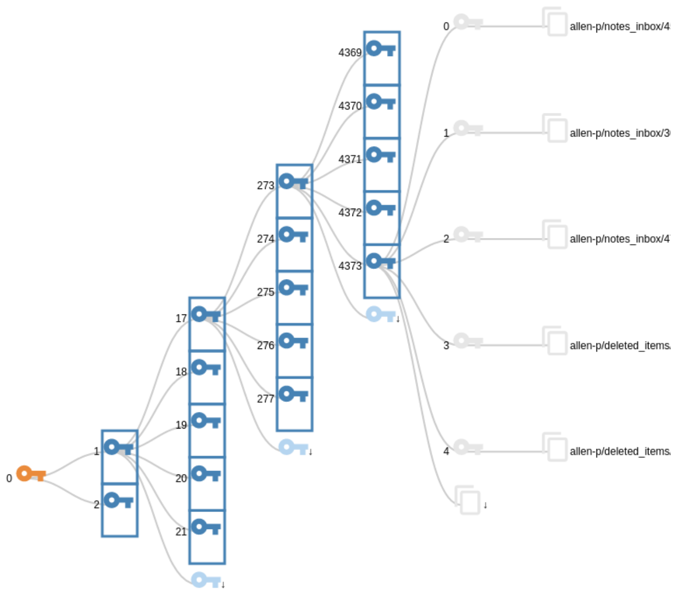


Figure 4: Screenshot of the interactive visualization for the key management data structure. Key 0 is secured by the TPM.

3. THE MICRO CONTENT MANAGEMENT SYSTEM (MCM)

In this demo, we present our proof-of-concept and bench marking application MCM[1]. MCM's functionality is based on on-premise Enterprise Content Management systems like IBM FileNet P8, but is designed to use outsourced cloud storage and client-side encryption. MCM stores objects and files inside storage containers in the Swift[2] object store. Whole containers can be transparently encrypted with a key-management mechanism that allows secure deletion of individual objects. MCM supports uploading and retrieving files, setting retention dates and scheduling deletion, extracting and viewing metadata, and analyzing and graphing analyses on this metadata. Our user interface also features interactive visualizations of the underlying key management data structures, which will be used in this Demo.

Figure 3 shows the high level architecture of MCM. The central component for this Demo is SDOS, the Secure Delete Object Store, which implements encryption as well as the key management for cryptographic deletion and also has the integration with the TPM.

We use three data management systems (bottom row of Figure 3): An Apache Kafka streaming platform for loosely coupled communication, an SQL database for storing and analyzing unencrypted metadata, and a Swift object store that holds all the encrypted data objects. Metadata is stored unencrypted in MCM because this allows us to execute queries on the cloud. The user can decide what metadata should be extracted from files, depending on their sensitivity. Cloud resources can then be used to query, search, filter, and analyze this metadata. File and container names are always stored as unencrypted metadata. If no further metadata is extracted, users must always retrieve (and decrypt) files before their content can be searched or analyzed locally.

We use the Swift REST-API protocol for our internal components in MCM, as shown by the blue lines in Figure 3. This protocol is used by the Swift object store and other large key/value stores (e.g. Ceph[3]) and their clients. Encryption and cryptographic deletion are handled by our SDOS component which is realized as such a Swift API-proxy. This enables us to use any unmodified Swift backend (e.g. SaaS) as well as any existing Swift clients. The API-proxies form a flexible pipeline. All MCM components can run multithreaded or distributed to enable horizontal scaling and high availability.

The Kafka streaming platform is used for triggering the execution of jobs for metadata extraction and replication as well as scheduled deletion of old objects. We use a relational database as a replicated metadata warehouse, as Swift lacks advanced querying capabilities for metadata (only retrieving and listing is possible). All the object metadata is primarily stored in Swift and then replicated to the RDBMS for analysis.

The location where the components from Figure 3 run is critical to the security of the system. In order to guarantee the secure deletion property, the content of the stored objects must never leave a trusted environment in unencrypted form. The same must be guaranteed for the encryption keys. Our SDOS encryption uses a tree structure for key management of which only the root key must be kept secure. All other keys are stored encrypted on Swift together with the data objects.

One possible separation of trusted/untrusted environment is given by the red line in Figure 3. It shows that all the data storage system can be outsourced to the public (untrusted) environment, because all sensitive data are encrypted. The master key for our key management mechanism is stored in, and never leaves the TPM.

[1] https://github.com/timwaizenegger/mcm-sdos
[2] http://docs.openstack.org/developer/swift/
[3] http://docs.ceph.com/docs/jewel/radosgw/swift

4. TRUSTED PLATFORM MODULES (TPM)

Trusted Platform Modules are a type of hardware security module that is used in PCs, Laptops, and Servers. They are already ubiquitous in those devices today and will achieve even higher spread in the future since Microsoft now lists a TPM as a mandatory hardware requirement for its Windows 10 operating system[4]. TPMs implement a specification by the Trusted Platform Group that defines core capabilities and security requirements [9]. TPMs generally contain a small storage as well as processing unit inside a tamper resistant physical package. Their most important feature is that certain areas of their storage unit can only be accessed by the internal processor. This means that some encryption keys can never leave the TPM but can only be used to de/encrypt data that is loaded into the TPM. TPMs furthermore have the capability to securely delete stored keys, and replace them with newly generated ones. This provides a secure basis for cryptographic deletion since a key is positively unrecoverable if it never leaves the TPM and is securely deleted inside the TPM later on.

The TPM's intended purpose is to support local disk and data encryption, as well as verified device identification. For data encryption, an encryption key is stored on disk but encrypted with the TPM master key. This encryption key can only be used after it was decrypted by an authenticated TPM. The actual data de- and encryption is then done by the main CPU, only de and encryption of the key is done inside the TPM. For device identification, TPMs contain an "endorsement key" that was signed by a trusted manufacturer master key. Remote services can challenge the TPM and verify the endorsement key in order to identify a certain machine. This is used for enterprise asset tracking as well as licence management for digital media (digital rights management).

TPMs can be used in custom applications with the limitation that no custom code can be run inside the TPM. Only the basic cryptographic operations are supported by the processor inside the TPM [11]. TPMs offer physical security and tamper resistance and can be used to secure master keys to custom cryptographic applications.

In MCM we use a TPM in order to store the master key on the SDOS core component. This master key (Key 0 in Figure 4) encrypts the first level of keys in a tree. The master key never leaves the TPM and is only used to en/decrypt the first level of the tree by loading this node into the TPM for processing.

5. DEMO OVERVIEW

This Demo will present a working prototype of a cloud storage system that offers transparent encryption with cryptographic deletion. We will show the theory behind our key-management mechanism (Key-Cascade), present the architecture of the cloud storage system, and demonstrate the integration with a Trusted Platform Module.

In our demo scenario we will first explain the layout of the system and the physical location of the individual components. We will then create new data containers with and without cryptographic deletion and show data ingestion and retrieval with different client applications. We will show how the encryption keys for new objects are generated and how they fit into the hierarchical Key-Cascade. We present the operations on the Key-Cascade including cryptographic deletion, show the capacity of the data structure as well as its scaling behavior. We then show how the Trusted Platform Module is integrated with the Key-Cascade and operations.

In this demo, the audience will learn about cryptographic deletion and its application to practical storage systems. Our integration of Trusted Platform Modules is relevant for applications outside of cryptographic deletion as well. Any system that employs cryptography can increase certain security aspects by integrating a TPM. Therefore, this demo is also relevant for researchers working in other areas of applied cryptography. Finally, our solution makes heavy use of the Swift object store and its REST-API which makes this demo relevant for researchers interested in Swift as well.

Screenshots of the user interface, all the application code, as well as more details about their capabilities, can be found on our Github page: https://github.com/timwaizenegger/mcm-bluebox

6. REFERENCES

[1] Amazon glacier with vault lock, SEC 17a-4(f) & CFTC 1.31(b)-(c) compliance assessment. Technical report, Cohasset Associates Inc., Aug. 2015.

[2] Regulation (EU) 2016/679: General data protection regulation. Technical report, European Parliament and Council, 2016.

[3] J. Barney, D. Lebutsch, C. Mega, S. Schleipen, and T. Waizenegger. Deletion of content in digital storage systems, March 2016. US Patent 9,298,951.

[4] S. Beresford. Deletion of records from national police systems. Technical report, UK National Police Chiefs' Council, May 2015.

[5] D. Brown, C. Arend, and A. Venkatraman. EU data protection reform will drive growth in european security and storage markets. *IDC ESS02X*, Oct. 2015.

[6] T. Conde. To delete or not delete - that's the question: A company's obligations to preserve records under the new electronic discovery rules. Technical report, Stoel Rives LLP, 2009.

[7] S. M. Diesburg and A.-I. A. Wang. A survey of confidential data storage and deletion methods. *ACM Comput. Surv.*, 43(1):2:1–2:37, Dec. 2010.

[8] S. Garfinkel and A. Shelat. Remembrance of data passed: a study of disk sanitization practices. *Security Privacy, IEEE*, 1(1):17–27, Jan. 2003.

[9] Trusted Computing Group. TPM 1.2 protection profile, 2016.

[10] T. Waizenegger. Poster presentation: SDOS: Secure deletion in the swift object store. In C. Nikolaou and F. Leymann, editors, *Proceedings of the 9th Symposium and Summer School On Service-Oriented Computing*, volume RC25564 of *IBM Research Report*. IBM, Dec. 2015.

[11] V. J. Zimmer, S. R. Dasari, and S. P. Brogan. Tcg-based firmware, white paper by Intel corporation and IBM corporation trusted platforms, 2009.

[4] https://msdn.microsoft.com/en-us/library/windows/hardware/dn915086(v=vs.85).aspx

Come and crash our database! – Instant recovery in action

Caetano Sauer
TU Kaiserslautern, Germany
csauer@cs.uni-kl.de

Gilson Souza
TU Kaiserslautern, Germany
gsantos@rhrk.uni-kl.de

Goetz Graefe
Google, Madison, WI, USA
goetzg@google.com

Theo Härder
TU Kaiserslautern, Germany
haerder@cs.uni-kl.de

ABSTRACT

We present a demonstration of instant recovery, a family of techniques to enable incremental and on-demand recovery from different classes of failures in transactional database systems. In contrast to traditional ARIES-based algorithms, instant recovery allows transactions to run concurrently to recovery actions—not only permitting earlier access to data that requires recovery but also using the post-failure access pattern to actually guide the recovery process. This mechanism prioritizes data needed most urgently after a failure, thus dramatically reducing the mean time to repair perceived by any individual transaction.

We have implemented instant recovery in an open-source storage manager and developed a Web-based interface to showcase its recovery capabilities. Users of this demo application are able to control the execution of various benchmarks and inject different types of failures arbitrarily, observing the system behavior and recovery progress live in a dashboard utility. Furthermore, since traditional ARIES recovery is also implemented, users can select the type of recovery and obtain a live graphical comparison of the different techniques.

1. INTRODUCTION

Database availability is a key challenge of scalable and reliable information systems. Improvements in availability can be achieved on two main fronts: increasing mean time to failure (MTTF) and decreasing mean time to repair (MTTR). Large businesses and Internet-scale services have invested heavily on the first front with highly redundant hardware configurations. The latter front, however, has not seen substantial improvements in the last decades, especially when considering the algorithms for logging and recovery in transactional database systems.

The vast majority of commercial and open-source database systems rely on techniques similar in essence to the ARIES family of recovery algorithms [5]. While ARIES works very well on traditional disk-based architectures with moderate transaction throughput and limited main-memory capacity, modern hardware and the systems designed to fully exploit its potential reveal severe limitations of traditional logging and recovery. These limitations include: long time to repair due to inefficient access patterns during recovery, inability to incrementally recover and enable access to most important data first, and high overhead on normal transaction processing.

Instant recovery was proposed recently as an alternative to ARIES that addresses its limitations on both traditional and modern hardware scenarios. Instead of being restricted to system failures and restart, it is designed to recover from all classes of failures known in the database literature. Furthermore, like ARIES, it relies on write-ahead logging with physiological log records, which means it can be incrementally implemented on an ARIES system, retaining all its capabilities while eliminating its limitations. For further elaboration on the limitations of previous techniques, the contribution of instant recovery, and empirical evaluation of the techniques, we refer to previous publications [1, 2, 7, 8].

This paper describes an interactive demo application designed to showcase the benefits of instant recovery in comparison with ARIES. After a brief overview of instant recovery in Section 2 below, the architecture and functionality of the demo application is described in Section 3. We provide a high-level description of the application and its interaction with the underlying transactional system, focusing on the user interaction and what attendees of the demonstration can expect to see. Finally, Section 4 provides a summary and some concluding remarks.

2. INSTANT RECOVERY IN A NUTSHELL

Instant recovery [1] is a family of algorithms designed to address different classes of failures in transactional systems. Table 1 summarizes such failure classes and their typical causes and effects. Details of the specific recovery mechanisms employed for each class are beyond the scope of our demonstration and have been explored extensively in previous research [1, 2, 7, 8]. This section discusses the fundamental characteristic common to all instant recovery techniques: the support for recovery actions that are executed concurrently to normal processing, provide incremental access to already-recovered data items, and exploit workload access patterns to guide recovery and prioritize data needed most

Series ISSN: 2367-2005

10.5441/002/edbt.2017.68

Failure class	Loss	Typical cause	Response
Transaction	Single-transaction progress	Deadlock	Rollback
System	Server process (in-memory state)	Software fault, power loss	Restart
Media	Stored data	Hardware fault	Restore
Single page	Local integrity	Partial writes, wear-out	Repair

Table 1: Failure classes, their causes, and effects (from [8])

urgently. In order to summarize these techniques and provide an overview of how instant recovery works, we discuss the case of restart after a system failure.

When a system failure occurs, the in-memory state of the database server process is lost and must therefore be recovered when the system comes back up. As in ARIES recovery, this boils down to determining which transactions must be rolled back in the UNDO phase (i.e., *active transactions*) and which pages must have their updates replayed in the REDO phase (i.e., *dirty pages*). Such information is collected with a sequential log scan in the *log analysis* phase, which covers the interval from the most recent checkpoint up to the last persisted log record.

In essence, the key characteristic that distinguishes instant restart from ARIES restart is that, once log analysis information is collected, the REDO and UNDO phases can be performed on a per-page and a per-transaction basis, respectively. In other words, recovery actions can be scheduled according to a variety of policies, and not just the schedule dictated by a sequential REDO or UNDO log scan. This means that as soon as log analysis is complete (which usually only takes a few seconds), transactions can immediately be admitted to the system and, as soon as they touch a page in need of recovery (i.e., a page marked dirty during log analysis) or incur a lock conflict with a transaction in need of rollback (i.e., a transaction marked active during log analysis), that single page or that single transaction can be recovered on demand. The basic design features that enable such on-demand recovery are (i) a per-page chain of log records that is also the basis of single-page recovery [2]; and (ii) tracking acquired locks during checkpoints and log analysis.

Fig. 1 illustrates the process of on-demand recovery during instant restart. In this scenario, log analysis has detected two dirty pages, A and B, whose expected page LSN is x and y, respectively. Furthermore, two pre-failure active transactions, T_1, holding locks with identifiers b and d, and T_2, with lock on f, were also detected. After log analysis, a new transaction is initiated—shown here as the gray box on the top left corner. This transaction attempts to fix page B in the buffer pool; because this page is marked as needing recovery, on-demand REDO processing kicks in and log records pertaining to this page are replayed by following a backward chain of log records (which is exactly the same process employed for single-page recovery [2]). Log replay of this single page happens concurrently with replay of any other page; in fact, it may even happen with asynchronous, ARIES-style REDO based on a forward log scan.

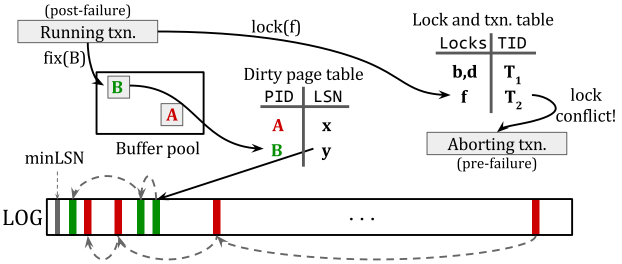


Figure 1: On-demand recovery in instant restart

After page B is restored, the new transaction attempts to acquire a lock on f. In this case, the lock table yields a conflict with transaction T_2; since this transaction is a pre-failure one, its rollback is triggered by this lock conflict. Because rollback of a pre-failure transaction works exactly in the same way as a transaction abort during normal processing (e.g., due to deadlock), the same logic is applied. Once such rollback is completed, the locks held by T_2 are released and the lock is finally granted to the waiting post-failure transaction.

The on-demand and incremental recovery schedules essentially reduce the MTTR as perceived by a single transaction by multiple orders of magnitude. ARIES restart recovery, in contrast, usually requires at least a full REDO log scan—which is typically the longest phase of recovery by far [6]—before the first post-failure transaction can complete[1].

In the next section, we describe the demo application proposed by this paper. It is the first interactive user interface ever developed for instant recovery, which can not only reproduce some of our empirical measurements, but also allow the user to interact with the workload, observe the system behavior graphically, and, most importantly, inject failures arbitrarily.

3. DEMONSTRATION

Instant recovery techniques such as single-page recovery, instant restart, and instant restore have been implemented over the past three years in the Zero [2] storage manager prototype, which is a fork of the well-known Shore-MT [3][3]. Zero has also been incorporated into the MariaDB database system (a modern fork of MySQL) as a storage engine module that can be used as an alternative to the popular InnoDB engine.

Our demonstration will provide a hands-on experience to interact with these systems under a variety of benchmark workloads. Focusing mainly on the logging and recovery aspects, the demo program enables users to inject different types of failures in a database workload running on Zero and observe the recovery process live in an intuitive graphical interface. Combined with a choice of parametrized workloads, this rich interface allows users to interactively explore

[1] Improvements to the ARIES algorithm aimed at enabling earlier access during recovery have been proposed [5, 4], but, in summary, none of them provides fully on-demand and incremental recovery to fine-granular objects, especially those needed most urgently by the application. The limitations of these "extended" versions of ARIES are also discussed in previous work [1, 8]

[2] https://github.com/caetanosauer/zero

[3] https://sites.google.com/site/shoremt/

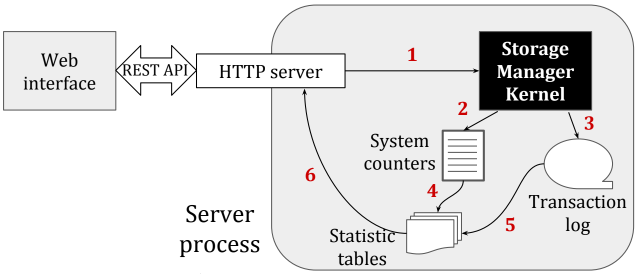


Figure 2: Architecture of the demo program

the potential of instant recovery techniques in improving database reliability.

To demonstrate the various instant recovery techniques and compare them with traditional ARIES-based recovery, we have developed a Web-based interface to control the execution of benchmarks and display relevant system statistics in a graphical way. This section describes how our demo program is organized, how it achieves these goals, and how the user can interact with it.

3.1 Demo program architecture

The diagram in Fig. 2 illustrates the main components of the demo program and their interaction. The Web interface serves two main purposes: first, it sends commands to the Zero storage manager (which may be running either independently or as a storage engine inside MariaDB) to control the execution of workloads as well as adjust system and benchmark parameters dynamically; and second, it serves as a "dashboard" to visualize real-time statistics.

The communication between Web client and server is implemented with a JSON-based REST API. Commands are available to start and stop a certain benchmark workload and—especially for the purposes of instant recovery—inject system crashes and persistent storage failures. System crashes cause the immediate destruction of all in-memory data structures of the server process—most importantly buffer pool, transaction manager, and lock table. In Fig. 2, the process of receiving a command via the REST API and forwarding it to the storage manager kernel is illustrated by arrow number 1. The following paragraphs discuss the remainder of the demo program architecture by referring to the numbered arrows.

While the storage manager kernel processes transactions, it generates two types of information which are relevant for the demo. First, a collection of system counters (arrow 2) is used to keep track of system events for which only the total number of occurrences is of interest. For example, counters of transaction commits, page reads and writes, log volume generated, number of active transactions, number of dirty pages, etc. are typical measures of interest.

The second collection mechanism is through the transaction log (arrow 3). By continuously analyzing the logs and performing various aggregations, events can be collected in a time-dependent manner, allowing more detailed statistics than those provided by simple counters. The information collected from the system counters and the transaction log is stored in a collection of statistic tables (arrows 4 and 5), which are finally serialized into the JSON format for display in the demo (arrow 6).

3.2 Visualizing the recovery process

As mentioned above, the demo application provides commands to inject failures into the running workload. We support injection of three of the four classes summarized in Table 1: system, media, and single-page failures. The remaining class—transaction failure—is not supported because there is not much to demonstrate in that case, as the abort of a single transaction does not cause any noticeable impact on system behavior.

When injecting a failure, the user may also choose which recovery method to employ: traditional ARIES or instant recovery. In addition to system failures, users can also inject a failure on persistent storage (e.g., single-page or whole-device failures) independently of system failures—i.e., different failure modes can be mixed and matched freely. Furthermore, failures can be re-injected at any time during recovery from a previous failure, demonstrating the independence and idempotency of recovery modes.

Regardless of whether recovery activities are being carried on or not, the dashboard of the demo program constantly displays statistics collected from the statistic tables described above, some of which can be selected for plotting. For instance, the user can observe the transaction throughput along with the buffer pool hit ratio. During instant restart after a system failure, these values should gradually rise as on-demand, incremental recovery progresses.

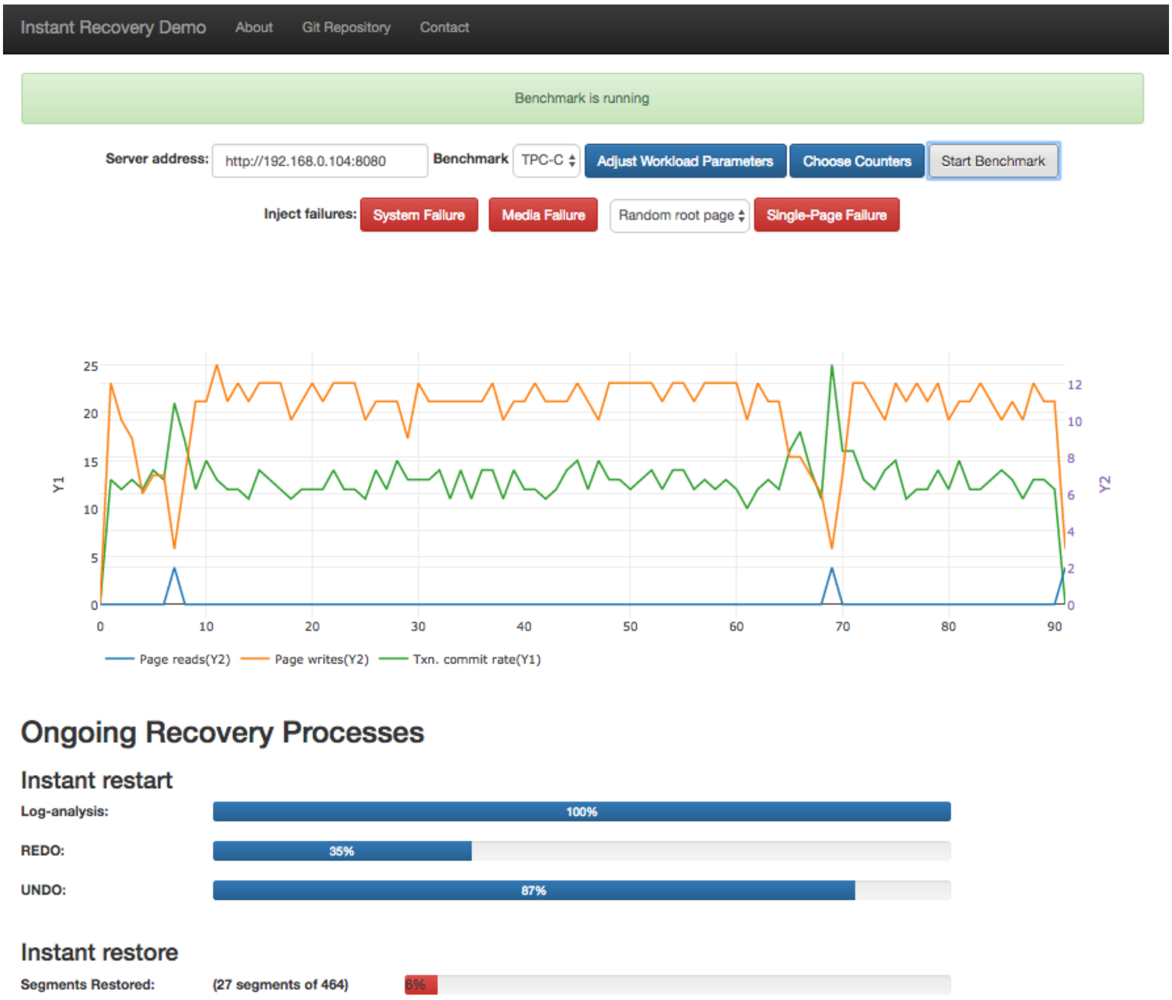
In addition to statistics available during normal processing, special statistics are collected and displayed during the recovery process—for instance, progress bars indicate the percentage of completion of each recovery phase (log analysis, REDO, and UNDO).

3.3 Screenshot walk-through

Fig. 3 shows a screenshot of our demo application in which a TPC-C workload is running and two recovery processes are currently active: instant restart (from a system failure) and instant restore (from a media failure). At the top, the IP address of the server process is provided along with the chosen workload—in this case TPC-C. An additional button opens up a pop-up window in which system and workload parameters can be adjusted—some of which can also be changed while the benchmark runs. Below that, three red buttons are provided to inject a system, media, or single-page failure. In the latter case, the user can additionally specify what kind of page should be selected for failure (in this case, a root page of an index). These buttons remain available even when recovery is already being carried out, allowing users to simulate failure-on-failure scenarios.

The middle part of the screenshot shows the graphical display component, in which three statistics are currently selected—transaction commit rate, page reads, and page writes. Using the "Choose counters" button on the top, users can select different statistics to plot. A text output of all stats and counters is also available but omitted here.

Finally, the bottom part shows the progress of currently ongoing recovery processes. In this case, a system restart is being carried on, and the progress bars show how much of the REDO and UNDO phases has been completed. Below that, a single progress bar displays the progress of a concomitant restore process, in which segments of the failed device are also restored incrementally. Note that transactions are running despite the ongoing recovery—this is the crucial feature of instant recovery.



Benchmark is running

Server address: http://192.168.0.104:8080 Benchmark TPC-C

Inject failures: Random root page

Ongoing Recovery Processes

Instant restart

Log-analysis: 100%

REDO: 35%

UNDO: 87%

Instant restore

Segments Restored: (27 segments of 464) 6%

Figure 3: Screenshot of the Instant Recovery Demo

4. SUMMARY AND CONCLUSIONS

Instant recovery drastically improves database system availability in the presence of failures, allowing incremental and on-demand recovery. Using a prototype transactional storage manager, we aim to demonstrate how these new recovery techniques behave in a real system. The demo application proposed here will enable attendees at the conference to interactively inject failures into a running workload and observe the instant recovery process live. The dashboard utility constantly collects relevant statistics from the database server process and displays them graphically according to customized selections made by the user. The Zero storage manager on which the demo is based is the first—and so far only—implementation of instant recovery. Therefore, the demo application will also provide a novel experience to most attendees.

5. REFERENCES

[1] G. Graefe, W. Guy, and C. Sauer. *Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, Media Restore, and System Failover, Second Edition*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2016.

[2] G. Graefe and H. A. Kuno. Definition, Detection, and Recovery of Single-Page Failures, a Fourth Class of Database Failures. *PVLDB*, 5(7):646–655, 2012.

[3] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proc. EDBT*, pages 24–35, 2009.

[4] C. Mohan. A cost-effective method for providing improved data availability during DBMS restart recovery after a failure. In *Proc. VLDB*, pages 368–379, 1993.

[5] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[6] C. Sauer, G. Graefe, and T. Härder. An empirical analysis of database recovery costs. In *RDSS (SIGMOD Workshops)*, 2014.

[7] C. Sauer, G. Graefe, and T. Härder. Single-pass restore after a media failure. In *Proc. BTW, LNI 241*, pages 217–236, 2015.

[8] C. Sauer, G. Graefe, and T. Härder. Instant restore after a media failure. Under submission, 2016.

μTOP: Spatio-Temporal Detection and Summarization of Locally Trending Topics in Microblog Posts

Paras Mehta
Freie Universität Berlin
Germany
paras.mehta@fu-berlin.de

Manuel Kotlarski
Freie Universität Berlin
Germany
kotlarski@inf.fu-berlin.de

Dimitrios Skoutas
IMIS, Athena R.C.
Greece
dskoutas@imis.athena-innovation.gr

Dimitris Sacharidis
Technische Universität Wien
Austria
dimitris@ec.tuwien.ac.at

Kostas Patroumpas
IMIS, Athena R.C.
Greece
kpatro@imis.athena-innovation.gr

Agnès Voisard
Freie Universität Berlin
Germany
agnes.voisard@fu-berlin.de

ABSTRACT

User-generated content in social media can offer valuable insights into local trends, events, and topics of interest. However, navigating through the vast amounts of posts either to retrieve certain pieces of information or to obtain an overview of the existing content, is often a challenging and overwhelming task. In this work, we present μTOP, a system for detecting and summarizing locally trending topics in microblog posts based on spatial, temporal and textual criteria. Using a sliding window model over an incoming stream of posts, μTOP detects locally trending topics, and associates each one with a spatio-temporal footprint. Then, for each spatial region and time period in which a certain topic is trending, the system generates a summary of the relevant posts, by selecting top-k posts based on the criteria of coverage and diversity. μTOP includes a Web-based user interface, providing a comprehensive way to visualize and explore the detected topics and their spatio-temporal summaries via a map and a timeline. The functionality of the system will be demonstrated using a continuously updated dataset containing more than 30 million geotagged tweets.

1. INTRODUCTION

Millions of posts are generated daily by users in social media, including text messages, photos, location check-ins, etc. These posts comprise textual content (typically, short text messages or tags), temporal information (the post's timestamp), and often spatial information (the post's geolocation). These spatial-temporal-textual objects are valuable pieces of information for revealing insights and trends regarding topics and events the users are interested in. However, given the sheer volume of this content, and its inherent redundancy and noise, retrieving relevant information or browsing and obtaining an overview of what is happening, is often a challenging and overwhelming task.

One solution to restrict the amount of incoming posts and focus on more relevant information is to filter out posts according to specified textual, spatial and/or temporal filters, as for example in publish/subscribe systems (e.g., [6]). However, given that social media content often involves new and emerging topics and events, the user may not know in advance what is interesting or relevant, and thus may not be able to specify a suitable geographic area, time period, or keywords for search.

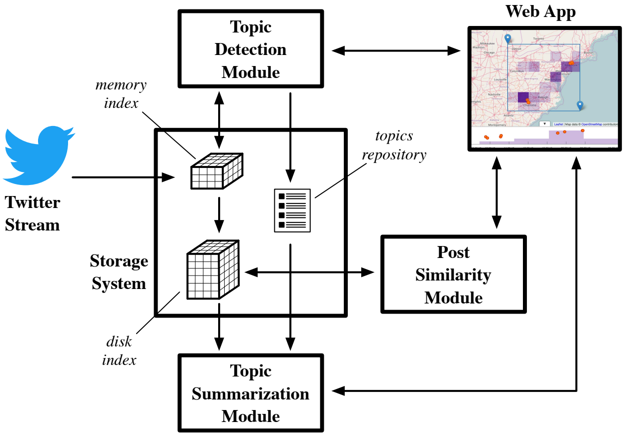


Figure 1: Architecture of μTOP.

To make it easier for users to get a quick grasp of the most important or interesting information, a common practice is to detect and present to the users a set of popular or trending topics (e.g., sets of hashtags in Twitter) that have high frequency (overall, or currently with respect to the past). However, the popularity of a topic is often not uniformly distributed across space and time; instead, a given topic may only be popular within specific geographic regions and over certain periods of time. In fact, recently there has been a lot of interest in finding local topics and events in Twitter (e.g., [1, 2, 3]). Nevertheless, even if a topic is detected as popular or trending, the posts belonging to it may still be in the order of hundreds or thousands. Hence, besides topic detection, generating topic summaries is also of high importance.

In this work, we present μTOP, a system for detecting and summarizing *locally trending topics* in streams of microblog posts. Each topic is represented by a set of one or more keywords (e.g., hashtags

Series ISSN: 2367-2005

10.5441/002/edbt.2017.69

in the case of Twitter), and is associated with a *spatio-temporal footprint*, i.e., a set of geographic regions and time periods over which this topic is identified to be popular. Thus, the spatio-temporal evolution of each detected topic is explicitly captured, and can be further explored. In fact, for each of these spatial regions and time intervals for which a topic is popular, μTOP can generate a summary of relevant tweets to describe the topic in more detail.

The discovery of locally trending topics is based on the approach presented in [5]. This method segments the space into a uniform grid and detects a set of trending topics in each cell by processing the incoming stream of posts applying a sliding window model. Thus, the topics are generated and monitored across space and time as new posts arrive and old ones expire, resulting in an evolving spatio-temporal footprint for each identified topic. Moreover, given a topic and its footprint, the system can generate a summary of relevant tweets. For this purpose, the relevant tweets are first retrieved using a spatial-temporal-textual filter, and then the top-k ones are selected according to the criteria of *coverage* and *diversity*, following the approach presented in [4].

Figure 1 presents an overview of the system architecture, which comprises the following main components. The *storage system*, detailed in Section 2, is responsible for ingesting the microblog posts (e.g., from Twitter's streaming API), and storing them in main memory and later on disk. In addition, this system maintains all topics and their spatio-temporal footprints. The core components of μTOP are the three data processing modules: *Topic Detection*, *Topic Summarization*, and *Post Similarity*, which are discussed in Section 3. Finally, the *Web App*, presented in Section 4, consists of the web-based user interface that allows users to issue queries, via invoking the appropriate modules, and visualize their results.

In the following sections, we describe in more detail the subsystems of μTOP, and present some usage examples in Section 5.

2. STORAGE SYSTEM

Each ingested post is represented as a spatial-temporal-textual object $D = \langle u, loc, t, \Psi \rangle$, where u is the identifier of the user making the post, $loc = (x, y)$ is the post's geolocation, t is the post's timestamp, and Ψ is a set of keywords representing the post's textual content.

To allow for efficient real-time detection of locally trending topics and the exploration (retrieval, summarization) of past topics and posts, we adopt a hybrid data indexing structure, involving both the main memory and the disk. This structure, depicted in Figure 2, indexes along all four attributes, latitude, longitude, time, and text. A 3-dimensional grid provides access along the first three attributes, while within each cell an inverted index provides efficient retrieval by keyword.

Each grid cell has size $g \times g \times \beta$, where g is a fixed arc range (for latitude and longitude) partitioning the world (or the spatial area of interest), and β is a fixed time interval. The inverted index of each cell associates each keyword with a list of posts in that cell that contain it. A slice of the grid in the temporal dimension containing posts that were published in an interval of β time units (e.g., one hour) is called a *pane*. The pane collecting the most recent posts is called the *head pane*.

The main memory index only stores the latest ω/β panes, and thus indexes posts that were published within a *sliding window* of ω time units (e.g., one day) in the past. This part of the grid is used by the topic detection module (Section 3.2). On the other hand, the disk-based index stores all panes except the head. This index is used by the topic summarization and the post similarity modules (Sections 3.3 and 3.4).

Besides this hybrid index structure, the storage system of μTOP includes a repository archiving all trending topics, along with their spatio-temporal footprints. The repository receives the continuous output of the topic detection module, and provides input to the topic summarization module when requested.

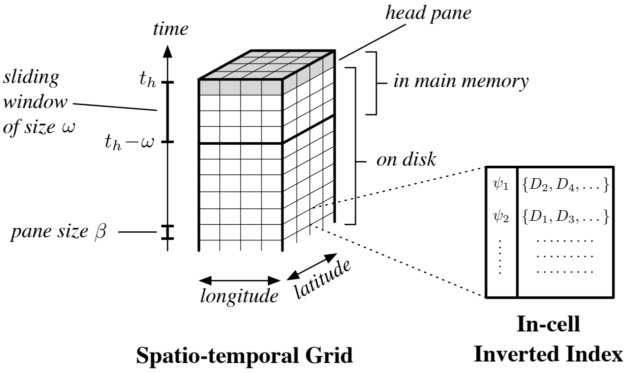


Figure 2: Overview of indexing scheme in μTOP.

3. SYSTEM MODULES

3.1 Preliminaries

First, we need to define textual, spatial and temporal distance functions between posts. Given two posts D_i and D_j, their *textual distance* δ_ψ is measured by the Jaccard similarity between their keyword sets:

$$\delta_\psi(D_i, D_j) = 1 - \frac{|D_i.\Psi \cap D_j.\Psi|}{|D_i.\Psi \cup D_j.\Psi|}.$$

The spatial and temporal distances are measured, respectively, by the Euclidean distance d of the posts' locations and the time difference of the posts' timestamps. To be able to aggregate distance scores across dimensions, we normalize spatial and temporal distances to values in the range $[0, 1]$ (notice that $\delta_\psi \in [0, 1]$). For that purpose, we assume that the posts under consideration are enclosed by a bounding box with diameter length γ and a time interval of length τ. Then, we define the (normalized) *spatial distance* δ_s and *temporal distance* δ_t as follows:

$$\delta_s(D_i, D_j) = \frac{d(D_i.loc, D_j.loc)}{\gamma}, \quad \delta_t(D_i, D_j) = \frac{|D_i.t - D_j.t|}{\tau}.$$

3.2 Topic Detection

In μTOP, *topic detection* is based on the work presented in [5]. We briefly describe the main aspects of the process below.

To process the incoming stream of posts, a lightweight, in-memory spatial index comprising a uniform spatial grid is used, as explained in Section 2. Upon arrival, each incoming post D is assigned to the corresponding grid cell c according to its geolocation $D.loc$. In each cell, the local stream of posts is processed to generate and maintain locally popular topics with respect to a sliding window W of range ω and sliding step β.

A topic C is characterized by a set of keywords (e.g., hashtags) $C.\Psi$ and is associated with the grid cell c and the time window W in which it is detected. The *popularity* $C.pop$ of a topic C within the cell c and time window W is determined by the number of users having posts in c and W that textually match this topic. We say that a post D matches a topic C if their textual similarity $\delta_\psi(D.\Psi, C.\Psi)$ is above a specified threshold $\theta_\psi \in [0, 1]$. The popularity score of a topic is normalized by the total number of users having posts within the cell c and window W. If an incoming post does not match any of the existing topics in the current cell and time window, a new topic is created having as keywords those appearing in this post. Eventually, those topics with popularity higher than a specified threshold $\theta_u \in [0, 1]$ are marked as *locally trending*, and are returned.

If the same topic is detected in multiple cells and/or time windows, these are merged to construct the topic's *spatio-temporal footprint* $C.\mathcal{F} = \{(c_i, W_i)\}$. Hence, this process not only detects locally popular topics but also explicitly associates each one with the exact geographic region(s) and time period(s) within which it was popular.

3.3 Topic Summarization

Once topics are detected, the next step is to get a summarized overview of each topic. A summary of a topic is already provided by the set of keywords defining it and its spatio-temporal footprint. However, a list of representative posts may also be needed in order to describe the topic in more detail.

For this purpose, μTOP can generate a summary, comprising k posts, for any part of the topic's spatio-temporal footprint. In other words, it can compute a set of k representative posts for any region and time window in which the given topic has been popular. The size of each summary, i.e., the value of the parameter k, can be specified by the user, and can be different for each summary.

The selection of the k representative posts to be included in the summary is based on the criteria of *coverage* and *diversity*. In particular, each summary is constructed by executing a *Coverage & Diversity Aware Top-k Spatial-Temporal-Keyword* (kCD-STK) query, following the approach presented in [4]. We outline the main aspects of this process next.

Formally, a kCD-STK query is defined by a tuple of the form $Q = \langle R, T, \Psi, k\rangle$, where R is a spatial region, T is a time interval, Ψ is a set of keywords, and k is the number of results to return. In our case, the filters R, T and Ψ are derived from the topic's keyword set and spatio-temporal footprint, while k is determined by the desired summary size. The distinguishing aspect of the kCD-STK query is that instead of selecting the top-k posts ranked by relevance, it selects a more representative set of k posts using the criteria of coverage and diversity, which are defined below.

Let $\mathcal{D}_F$ denote the set of all posts satisfying the spatial, temporal and textual filters R, T and Ψ in the query Q. The *coverage* of a post $D \in \mathcal{D}_F$ is defined as the ratio of relevant posts that are within spatial distance θ_s and temporal distance θ_t from D, i.e.:

$$cov(D, \mathcal{D}_F) = \frac{|\{D' \in \mathcal{D}_F : d_s(D, D') \leq \theta_s \wedge d_t(D, D') \leq \theta_t\}|}{|\mathcal{D}_F|}.$$

This is a measure of how representative this particular post is with respect to other relevant posts. Moreover, this is extended to measure the coverage of a set of selected posts $\mathcal{R} \subseteq \mathcal{D}_F$ of size k:

$$cov(\mathcal{R}, \mathcal{D}_F) = \frac{1}{k} \sum_{D \in \mathcal{R}} cov(D, \mathcal{D}_F).$$

Essentially, the criterion of coverage favors the selection of posts from locations that contain a large number of relevant posts.

On the other hand, to avoid a high degree of redundancy, the criterion of *diversity* is used to increase the dissimilarity among the selected posts. Specifically, the diversity of a pair of posts $D_i, D_j \in \mathcal{D}_F$ is defined as:

$$div(D_i, D_j) = \alpha \cdot d_s(D_i, D_j) + (1 - \alpha) \cdot d_t(D_i, D_j),$$

where $\alpha \in [0, 1]$ is an adjustable weight parameter between the spatial and the temporal distances. Furthermore, the diversity of a set of posts $\mathcal{R} \subseteq \mathcal{D}_F$ of size k is calculated as:

$$div(\mathcal{R}) = \frac{1}{k \cdot (k-1)} \sum_{D_i, D_j \in \mathcal{R}, i \neq j} div(D_i, D_j).$$

Based on the above, the kCD-STK query returns a set of k posts $\mathcal{R}^*$ that maximizes a combined measure of coverage and diversity:

$$\mathcal{R}^* = \operatorname*{arg\,max}_{\mathcal{R} \subseteq \mathcal{D}_F, |\mathcal{R}| = k} \{(1 - \lambda) \cdot cov(\mathcal{R}, \mathcal{D}_F) + \lambda \cdot div(\mathcal{R})\},$$

where $\lambda \in [0, 1]$ is a parameter determining the tradeoff between maximum coverage ($\lambda = 0$) and maximum diversity ($\lambda = 1$).

3.4 Retrieving Similar Posts

The above process provides a flexible and adjustable way to get a summary of representative and diverse posts for a topic across the whole extent of its spatio-temporal footprint. Then, the user can further drill down into the topic, by selecting any of the posts in the presented summary that seems interesting, and requesting other similar posts to it. That is, the posts contained in each summary can serve as *seeds* for further exploration of the topic's contents.

This is performed by executing a standard top-k spatial-temporal-keyword query $Q = \langle loc, t, \Psi, k\rangle$, where loc, t, and Ψ are, respectively, the location, the timestamp and the keyword set of the selected post D, and k is the number of similar posts to be retrieved. In this case, the query returns the top-k results ranked by *relevance* determined by an aggregate distance score δ combining the partial distance scores in the spatial, temporal and textual dimensions, i.e.:

$$\delta(D, D') = w_s \cdot \delta_s(D, D') + w_t \cdot \delta_t(D, D') + w_\psi \cdot \delta_\psi(D, D')$$

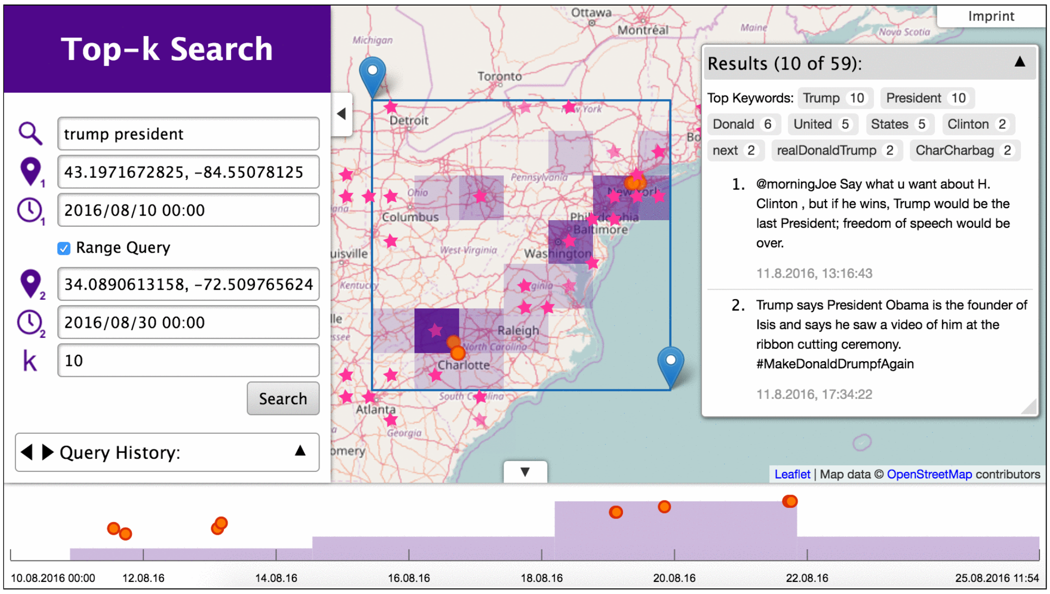
where $w_s \in [0, 1]$, $w_t \in [0, 1]$ and $w_\psi = 1 - w_s - w_t$ are weights determining the relative importance of each distance score.

4. USER INTERFACE

The user interface is shown in Figure 3. The map continuously depicts locally trending topics as discovered by the topic detection module. Topics are shown as stars, with brightness indicating popularity. Hovering over a star reveals the topic's spatial footprint, whereas clicking on it shows its keywords together with two options (Figure 4 left). The first option is to invoke the *post similarity* module to retrieve a ranked list of similar posts (in terms of spatial proximity, time closeness, and textual relevance). The resulting posts are displayed in a pop-up window on the right, and also as orange dots on the map and on the timeline located at the bottom.

The second option for a locally trending topic is to explore its spatio-temporal footprint by invoking the *topic summarization* module. The sidebar on the left displays a form detailing the spatial and temporal ranges for the summary, as well as the keywords and the number of returned results (default is ten). Naturally, the user can specify her own summarization request. The summarization results are listed in a pop-up window on the right, where the user can filter them by the top keywords shown at the top. The spatial and temporal distributions of the results are shown on the map and on a timeline at the bottom using orange bullets, respectively. The height of the purple bars in the timeline indicates the average coverage in the corresponding temporal range. Similarly, the purple rectangles on the map illustrate the average coverage in the corresponding regions. The darker the color, the higher the coverage in the area.

Further exploration of the topic summarization results is provided by two means. First, the timeline allows the user to filter the results by selecting a temporal sub-range. This issues a new topic summarization request and updates the results. Second, by clicking on a result on the map, besides showing its content and a link to



Top-k Search

trump president

43.1971672825, -84.55078125

2016/08/10 00:00

☑ Range Query

34.0890613158, -72.509765624

2016/08/30 00:00

k 10

Search

Query History:

Results (10 of 59):

Top Keywords: Trump 10 President 10 Donald 6 United 5 States 5 Clinton 2 next 2 realDonaldTrump 2 CharCharbag 2

1. @morningJoe Say what u want about H. Clinton , but if he wins, Trump would be the last President; freedom of speech would be over.
11.8.2016, 13:16:43

2. Trump says President Obama is the founder of Isis and says he saw a video of him at the ribbon cutting ceremony. #MakeDonaldDrumpfAgain
11.8.2016, 17:34:22

Figure 3: The user interface showing the results of a topic summarization request.

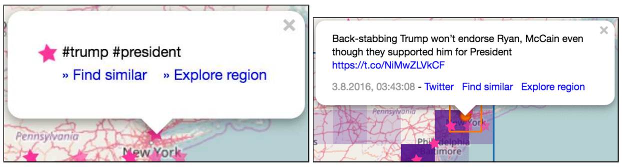
the post, μTOP displays two additional links (Figure 4 right). The one issues a *retrieve similar posts* request, while the other allows the user to further explore the highlighted spatio-temporal region issuing a new *topic summarization* request.

5. DEMONSTRATION

To demonstrate the efficiency and effectiveness of μTOP, tweets are continuously being collected from the public Twitter Streaming API[1]; the current dataset contains over 30 million geotagged tweets with worldwide coverage. The topics are monitored on a stream arriving at an average rate of approximately 500,000 tweets per day. A live demo[2] of μTOP is available online, accompanied by a video[3] explaining and demonstrating its functionality.

Next, we outline a typical usage scenario for demonstration. Initially, the user interface shows locally trending topics on a map, depicted by star icons. Clicking on a star icon reveals the topic's hashtags, for example "#trump #president", as shown in Figure 4. The *explore* link is then used to summarize the topic. It issues a topic summarization request that displays the resulting tweets in a list, on the map and on the timeline. Alternatively, the user may enter query parameters manually using the form in the sidebar on the left, for example to increase the spatial area and time interval.

At the top of the result list a set of keywords is shown that are popular among the result set. This reveals new keywords that are frequently used together with the query keywords *Trump* and *President*. For example *Clinton* is used in 20% of the results. We can click on it to view only those posts that contain this word.



#trump #president
» Find similar » Explore region

Back-stabbing Trump won't endorse Ryan, McCain even though they supported him for President
https://t.co/NiMwZLVkCF
3.8.2016, 03:43:08 - Twitter Find similar Explore region

Figure 4: A locally trending topic, and a post summarizing it.

When a topic is summarized, the average coverage is shown as purple blocks and bars in addition to the results. This allows to easily identify spatial regions and time intervals where the topic is popular. For example, Figure 3 shows that the topic is popular around New York City and between the 18th and 22nd of August. This spatial region and time interval can be further explored by issuing another topic summarization request, for example by moving the blue markers on the map or selecting a temporal range on the timeline. We can return to the previous result set by clicking the back-arrow button in the *Query History*, shown in the sidebar.

Instead of summarizing a particular topic, we can also explore a topic by invoking a post similarity search without limiting the spatial and temporal range. By clicking the *Find similar* link, a list of posts similar in spatial, temporal, and textual content is compiled.

Acknowledgements

This work was partially supported by the EU Project City.Risks (H2020-FCT-2014-653747).

6. REFERENCES

[1] H. Abdelhaq, C. Sengstock, and M. Gertz. EvenTweet: Online localized event detection from twitter. *PVLDB*, 6(12):1326–1329, 2013.

[2] C. Budak, T. Georgiou, D. Agrawal, and A. El Abbadi. Geoscope: Online detection of geo-correlated information trends in social networks. *PVLDB*, 7(4):229–240, 2013.

[3] W. Feng, C. Zhang, W. Zhang, J. Han, J. Wang, C. Aggarwal, and J. Huang. STREAMCUBE: hierarchical spatio-temporal hashtag clustering for event exploration over the twitter stream. In *ICDE*, pages 1561–1572, 2015.

[4] P. Mehta, D. Skoutas, D. Sacharidis, and A. Voisard. Coverage and diversity aware top-k query for spatio-temporal posts. In *SIGSPATIAL*, page 19, 2016.

[5] K. Patroumpas and M. Loukadakis. Monitoring spatial coverage of trending topics in twitter. In *SSDBM*, pages 7:1–7:12, 2016.

[6] X. Wang, Y. Zhang, W. Zhang, X. Lin, and Z. Huang. SKYPE: top-k spatial-keyword publish/subscribe over sliding window. *PVLDB*, 9(7):588–599, 2016.

[1] https://dev.twitter.com/streaming/public
[2] http://mtop.imp.fu-berlin.de
[3] https://youtu.be/OmXJUGndaQA

Insights into the Comparative Evaluation of Lightweight Data Compression Algorithms

Patrick Damme, Dirk Habich, Juliana Hildebrandt, Wolfgang Lehner
Database Systems Group
Technische Universität Dresden
01062 Dresden, Germany
{firstname.lastname}@tu-dresden.de

ABSTRACT

Lightweight data compression is frequently applied in in-memory database systems to tackle the growing gap between processor speed and main memory bandwidth. In recent years, the number of available compression algorithms has grown considerably. Since the correct choice of one of these algorithms requires understanding of their performance behavior, we systematically evaluated several state-of-the-art compression algorithms on a multitude of different data characteristics. In this demonstration, the attendee will learn our findings in an interactive tour through our obtained measurements. The most important insight is that there is no single-best algorithm, but that the choice depends on the data characteristics and is non-trivial.

1. INTRODUCTION

The continuous growth of data volumes is a major challenge for the efficient data processing. With the growing capacity of the main memory, efficient analytical data processing becomes possible [6]. However, the gap between computing power of the CPUs and main memory bandwidth continuously increases, which is now the main bottleneck for an efficient data processing. To overcome this bottleneck, data compression plays a crucial role [1, 11]. Aside from reducing the amount of data, compressed data offers several advantages such as less time spent on load and store instructions, a better utilization of the cache hierarchy, and less misses in the translation lookaside buffer.

This compression solution is heavily exploited in modern in-memory column stores for efficient query processing [1, 11]. Here, relational data is maintained using the *decomposition storage model* [3]. That is, an n-attribute relation is replaced by n binary relations, each consisting of one attribute and a surrogate indicating the record identity. Since the latter contains only *virtual* ids, it is not stored explicitly. Thus, each attribute is stored separately *as a sequence of values*. For the lossless compression of sequences of values (in particular integer values), a large variety of lightweight algorithms has been developed [1, 2, 7, 8, 9, 10, 11][1]. In contrast to heavyweight algorithms, lightweight algorithms achieve comparable or even better compression rates. Moreover, the computational effort for the (de)compression is lower than for heavyweight algorithms. To achieve these unique properties, each lightweight compression *algorithm* employs one or more basic compression *techniques* such as frame-of-reference [11] or null suppression [1], which allow the appropriate utilization of contextual knowledge like value distribution, sorting, or data locality.

In recent years, the efficient *vectorized* implementation of these lightweight compression algorithms using SIMD (Single Instruction Multiple Data) instructions has attracted a lot of attention [7, 8, 9, 10], since it further reduces the computational effort. To better understand these vectorized lightweight compression algorithms and to be able to select a suitable algorithm for a given data set, the behavior of the algorithms regarding different data characteristics has to be known. In particular, the behavior in terms of *performance* (compression, decompression and processing) and *compression rate* is of interest. Therefore, we have done an experimental survey of a broad range of algorithms with different data characteristics in a systematic way. We used a multitude of synthetic data sets as well as two commonly used real data sets. While we have already published *selected* results of our exhaustive evaluation in [5], this demonstration makes use of our *entire* corpus of measurements, which we obtained from an even larger collection of data characteristics than we could discuss in [5]. More precisely, the goals of this demonstration are the following:

1. We present our experimental methodology. This includes our selection of data characteristics and algorithms as well as our benchmark framework [4].
2. We explain the employed implementations of the compression algorithms and thus provide background knowledge for understanding the empirical results.
3. We explore various visualizations of the results of our systematic evaluation using an interactive web-interface.
4. Finally, we provide detailed insights into the behavior of the considered lightweight compression algorithms depending on the properties of the uncompressed data.

The remainder of the paper is organized as follows: In Section 2, we provide some important background knowledge on the area of lightweight data compression. An overview of

[1] Without claim of completeness.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.70

our underlying systematic evaluation is given in Section 3. Finally, Section 4 describes what attendees will experience in our demonstration.

2. LIGHTWEIGHT DATA COMPRESSION

This section gives an overview of the basic concepts of lightweight data compression. We distinguish between compression *techniques* and compression *algorithms*, whereby each algorithm implements one or more techniques.

2.1 Techniques

There are five basic lightweight techniques to compress a sequence of values: frame-of-reference (FOR) [11], delta coding (DELTA) [7], dictionary compression (DICT) [1, 11], run-length encoding (RLE) [1], and null suppression (NS) [1]. FOR and DELTA represent each value as the difference to either a certain given reference value (FOR) or to its predecessor value (DELTA). DICT replaces each value by its unique key in a dictionary. The objective of these three well-known techniques is to represent the original data as a sequence of small integers, which is then suited for actual compression using the NS technique. NS is the most studied lightweight compression technique. Its basic idea is the omission of leading zeros in the bit representation of small integers. Finally, RLE tackles uninterrupted sequences of occurrences of the same value, so called *runs*. Each run is represented by its value and length. Hence, the compressed data is a sequence of such pairs.

Generally, these five techniques address different data levels. While FOR, DELTA, DICT, and RLE consider the *logical* data level, NS addresses the *physical* level of bits or bytes. This explains why lightweight data compression algorithms are always composed of one or more of these techniques. The techniques can be further divided into two groups depending on how the input values are mapped to output values. FOR, DELTA, and DICT map each input value to exactly one integer as output value (*1:1 mapping*). The objective of these three techniques is to achieve smaller numbers which can be better compressed on the bit level. In RLE, not every input value is necessarily mapped to an encoded output value, because a successive subsequence of equal values is encoded in the output as a pair of run value and run length (*N:1 mapping*). In this case, a compression is already done at the logical level. The NS technique is either a 1:1 or an N:1 mapping depending on the implementation.

2.2 Algorithms

The genericity of these techniques is the foundation to tailor the algorithms to different data characteristics. Therefore, a lightweight data compression algorithm can be described as a cascade of one or more of these basic techniques. On the level of the algorithms, the NS technique has been studied most extensively. There is a very large number of specific algorithms showing the diversity of the implementations for a single technique. The pure NS algorithms can be divided into the following classes [10]: (i) bit-aligned, (ii) byte-aligned, and (iii) word-aligned.[2] While bit-aligned NS algorithms try to compress an integer using a minimal number of *bits*, byte-aligned NS algorithms compress an integer with a minimal number of *bytes* (1:1 mapping). The word-aligned NS algorithms encode as many integers as possible into 32-bit or 64-bit words (N:1 mapping).

The logical-level techniques have not been considered to such an extent as the NS technique *on the algorithm level*. In most cases, they have been investigated in connection with the NS technique. For instance, PFOR-based algorithms implement the FOR technique in combination with a bit-aligned NS algorithm [11]. These algorithms usually subdivide the input in subsequences of a fixed length and calculate two parameters per subsequence: a reference value for the FOR technique and a common bit width for NS. Each subsequence is encoded using their specific parameters, thereby the parameters are data-dependently derived. The values that cannot be encoded with the given bit width are stored separately with a greater bit width.

3. SYSTEMATIC EVALUATION

The effective employment of these lightweight compression algorithms requires a thorough understanding of their behavior in terms of performance and compression rate. Therefore, we have conducted an extensive experimental evaluation of several lightweight compression algorithms on a multitude of different data characteristics. Furthermore, we have already published some of the results in [5]. In this section, we present the key facts about our systematic evaluation, which is the basis of the demonstration. Besides the considered algorithms and data characteristics, we also provide details on our experimental setup.

3.1 Considered Algorithms

Our selection of algorithms follows two principal goals: Firstly, all five techniques of lightweight data compression should be represented. Secondly, the implementations should reflect the state-of-the-art in terms of efficiency.

Regarding efficiency, the use of SIMD (Single Instruction Multiple Data) instruction set extensions such as Intel's SSE and AVX plays a crucial role. These allow the application of one operation to multiple elements of so-called vector registers at once. The available operations include parallel arithmetic, logical, and shift operations as well as permutations. These are highly relevant to lightweight compression algorithms. In fact, the main focus of recent research [7, 8, 9, 10] in this field has been the employment of SIMD instructions to speed up (de)compression. Consequently, most of the algorithms we evaluated make use of SIMD extensions.

Regarding the techniques, we consider both, algorithms implementing a single technique and cascades of one logical-level and one physical-level technique. Since implementations of the logical-level techniques are hardly available in isolation, i.e., without the combination with NS, we use our own *vectorized* reimplementions of RLE, DELTA, and FOR, and a *sequential* reimplementation of DICT. Concerning the physical-level technique NS, however, there are several publicly available high-quality implementations, e.g., the FastPFOR-library by Lemire et al.[3]. We used such available implementations whenever possible and reimplemented only the recently introduced algorithm SIMD-GroupSimple [10], since we could not find an implementation of it. Table 1 gives an overview of the NS algorithms in our systematic evaluation. Note that all three classes of NS are represented in this selection. Due to space limitations, we cannot elab-

[2] [10] also defines a *frame-based* class, which we omit, as the representatives we consider also match the *bit-aligned* class.

[3] https://github.com/lemire/FastPFOR

Class	Algorithm	Ref.	Code origin	SIMD
bit-aligned	4-Gamma	[9]	Schlegel et al.	yes
	SIMD-BP128	[7]	FastPFOR-lib[3]	yes
	SIMD-FastPFOR	[7]	FastPFOR-lib[3]	yes
byte-aligned	4-Wise NS	[9]	Schlegel et al.	yes
	Masked-VByte	[8]	FastPFOR-lib[3]	no/yes
word-aligned	Simple-8b	[2]	FastPFOR-lib[3]	no
	SIMD-GroupSimple	[10]	our own code	yes

Table 1: The considered NS algorithms.

orate further on these algorithms. Instead, we recommend to read [5], which contains high-level descriptions of these.

To enable the *systematic* investigation of combinations of logical-level and physical-level algorithms, we implemented a generic cascade algorithm, which can be specialized for *any* pair of compression algorithms. This cascade algorithm partitions the uncompressed data into blocks of a certain size and does the following for each block: First, it applies the logical-level algorithm to the uncompressed block storing the result to a small intermediate buffer. Second, it applies the physical-level algorithm to that intermediate buffer and appends the result to the output. The decompression works the opposite way. We choose the block size such that it fits into the L1 data cache in order to achieve high performance.

To sum up, we investigated 4 logical-level algorithms, 7 physical level algorithms, and 4 × 7 = 28 cascades, yielding a total of 39 algorithms. For each of these algorithms, we consider the compression and decompression part. Additionally, we implemented a summation of the compressed data for each algorithm as an example of data processing.

3.2 Considered Data Sets

We made extensive use of *synthetic* data, since it allows us to carefully vary all relevant data properties. More precisely, we experimented with various combinations of the total number of data elements, the number of distinct data elements, the distribution of the data elements, the distribution of run lengths, and the sort order. We employed random distributions which are frequently encountered in practice, such as uniform, normal, and zipf. Additionally we introduced different amounts of outliers. For each data set, we vary one of these properties, while the others are fixed. That way, we can easily observe the impact of the varied property. To give an example, one of our data sets consists of 100 M uncompressed 32-bit integers, 90% of which follow a normal distribution with a small mean, while 10% are normally distributed outliers for which we vary the mean.

Additionally, we employed two *real* data sets which are commonly used in the literature on lightweight compression: the postings lists of the GOV2 and ClueWeb09b document collections.

3.3 Experimental Setup

All algorithms are implemented in C/C++ and we compiled them with `g++` 4.8 using the optimization flag `-O3`. Our Ubuntu 14.04 machine was equipped with an Intel Core i7-4710MQ (Haswell) processor with 4 physical and 8 logical cores running at 2.5 GHz. The L1 data, L2, and L3 caches have a capacity of 32 KB, 256 KB and 6 MB, respectively. We use only one core at any time of our evaluation to avoid competition for the shared L3 cache. The capacity of the DDR3 main memory was 16 GB.

All experiments happened entirely in main memory. The disk was never accessed during the time measurements. We conducted the evaluation using our benchmark framework [4]. The synthetic data was generated by our data generator once per configuration of the data properties. During the executions, the runtimes and the compression rates were measured. To achieve reliable measurements, we emptied the cache before each algorithm execution (by copying an array much larger than the L3 cache) and repeated all time measurements 12 times, whereby we used the wallclock-time.

4. DEMONSTRATION SCENARIO

For our demonstration we will use an interactive web-interface (Fig. 1a). This interface is based on `jupyter`[4], a tool widely used for interactive scientific data processing. While we are able to present *how* we conducted our systematic evaluation using our benchmark framework [4], the main focus of the demonstration will be the exploration of the collection of measurements.

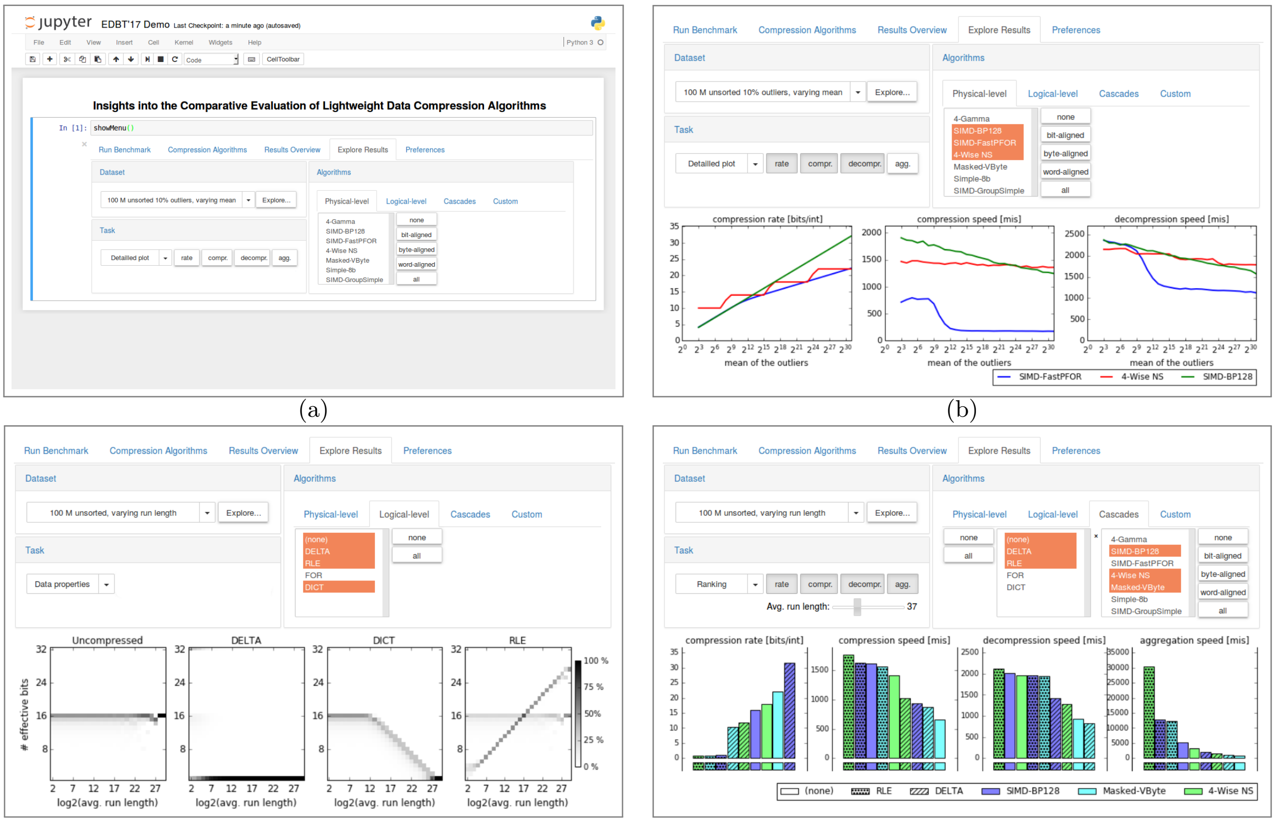
Our demonstration offers several opportunities for involving the attendee. He or she can select the algorithms to be compared as well as the data characteristics (Fig. 1a). Optionally, the attendee can define a trade-off between the possible optimization goals of lightweight compression. For instance, there are scenarios in which the decompression speed is most relevant, while the compression speed is not of interest, or in which the compression rate is more important than the speeds. Defining such a trade-off can help to determine the best algorithm for a given use case.

Regarding the available data sets, we already have a very large collection of measurements, which we obtained in our systematic evaluation as mentioned in Section 3.2. These reflect a multitude of combinations of relevant data properties. Nevertheless, if the attendee wishes so, he or she can also define a configuration of data characteristics, we have not considered so far. In this case, we would simply run the evaluation on the fly.

While it is possible to investigate any of the considered compression algorithms on any data in an interactive and spontaneous way, we would like to highlight the following prepared scenarios:

- *Impact of the data distribution on the physical-level algorithms* (Fig. 1b). In this scenario, the attendee will learn about the general behavior of null suppression algorithms depending on their class. Furthermore, he or she will find out, how different data distributions influence this behavior additionally.
- *Impact of the logical-level algorithms on the data characteristics* (Fig. 1c). Here, the attendee can observe how the application of purely logical-level algorithms such as RLE and FOR changes the properties of the underlying data. For this purpose, we employ, e.g., visualizations of the data distributions. We will discuss with the attendee, in how far these changed properties are suited for the following application of a physical-level algorithm.
- *Cascades of logical-level and physical-level algorithms* (Fig. 1d). Finally, the attendee will learn that the combination of logical-level and physical-level algorithms

[4] http://www.jupyter.org



(a)

(b)

(c)

(d)

Figure 1: Screenshots of our demonstration web-interface.

can yield significant improvements in terms of speed or compression rate, but not necessarily both. Defining a trade-off can help to make a final decision. Moreover, depending on the data, not all combinations are beneficial. The attendee will understand that even if the logical-level technique is fixed, the choice of the NS algorithm can make a significant difference.

By the end of the demonstration, the attendee will appreciate that there is no single-best lightweight compression algorithm, but that the choice depends on the data characteristics as well as the optimization goal and is non-trivial. At this point, the results of our systematic evaluation can help to select the best algorithm for a given data set.

Acknowledgments

This work was partly funded by the German Research Foundation (DFG) in the context of the project ”Lightweight Compression Techniques for the Optimization of Complex Database Queries” (LE-1416/26-1).

5. REFERENCES

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[2] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Softw., Pract. Exper.*, 40(2), 2010.

[3] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. *SIGMOD Rec.*, 14(4), 1985.

[4] P. Damme, D. Habich, and W. Lehner. A benchmark framework for data compression techniques. In *TPCTC*, 2015.

[5] P. Damme, D. Habich, and W. Lehner. Lightweight data compression algorithms: An experimental survey. In *EDBT*, 2017.

[6] T. Kissinger, T. Kiefer, B. Schlegel, D. Habich, D. Molka, and W. Lehner. ERIS: A numa-aware in-memory storage engine for analytical workloads. In *ADMS*, 2014.

[7] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.*, 45(1), 2015.

[8] J. Plaisance, N. Kurz, and D. Lemire. Vectorized vbyte decoding. *CoRR*, abs/1503.07387, 2015.

[9] B. Schlegel, R. Gemulla, and W. Lehner. Fast integer compression using SIMD instructions. In *DaMoN*, 2010.

[10] W. X. Zhao, X. Zhang, D. Lemire, D. Shan, J. Nie, H. Yan, and J. Wen. A general simd-based approach to accelerating compression algorithms. *ACM Trans. Inf. Syst.*, 33(3), 2015.

[11] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.

Multi-workflow optimization in PAW

Maxim Filatov
University of Geneva
maxim.filatov@unige.ch

Verena Kantere
University of Geneva
verena.kantere@unige.ch

ABSTRACT

As business decisions and strategies become more and more automated, real-time, and data-driven, enterprises need to create, manage and execute end-to-end analytics workflows that process increasing data volumes, from new heterogeneous data sources, on specialized processing engines. Designing and optimizing such workflows is a challenging task since they span a variety of systems and tools. To address these needs, we present the Platform for Analytics Workflows (PAW). PAW enables workflow design, execution, analysis and optimization with respect to time efficiency, over multiple execution engines and storage repositories. In this paper, we focus on the demonstration of the functionality of PAW related to multi-workflow optimization. We demonstrate the functionality of PAW for users with various expertise and its capabilities with respect to workflow analysis and optimization. We employ several scenarios of running workloads of workflows with and without PAW's optimization on real use cases and data from the telecommunication domain and web analytics, but also on synthetic use cases and data.

1. INTRODUCTION

The analysis of Big Data is a core and critical task in multifarious domains of science and industry. Such analysis needs to be performed on a range of data stores, both traditional and modern, on data sources that are heterogeneous in their schemas and formats, and on a diversity of query engines. Moreover, such analysis is also intensive and systematic. This means that many users access the same data at the same time with different or similar target results, and, such results are the output of an analytics process. Thus, a system that enables such analytics processes on Big Data needs to be able to manage several workflows and execute them in an optimal manner. Workflow execution can be extremely resource- and time-consuming. Therefore, the optimization of the execution of a single workflow but also of the joint execution of several workflows is very important for the efficiency of such a system.

Commercial Extract-Transform-Load (ETL) tools (e.g. [3], [2]) provide little support for automatic optimization. They provide hooks for the ETL designer to specify for example which flows may run in parallel or where to partition flows for pipeline parallelism. Some ETL engines such as PowerCenter [3] support PushDown optimization, which pushes operators expressed in SQL from the ETL flow down to the source or target database engine. The rest of the transformations are executed in the data integration server. The challenge of optimizing the entire workflow remains unsolved.

Towards this direction, HFMS [4] performs optimization and execution across multiple engines. Work related to HFMS [5] focuses on optimizing flows for several objectives: performance, fault-tolerance and freshness over multiple execution engines. HFMS uses many optimization strategies, such as parallelization, recovery points, function shipping, data shipping, decomposition, etc. However, HFMS does not focus on managing or optimizing in a joint manner multiple workflows.

We demonstrate a novel technique for multi-workflow optimization that is implemented as part of our system called PAW (Platform for Analytics Workflows), a platform for the design, analysis and execution of analytics workflows. To the best of our knowledge, there is no previous work on multi-workflow optimization. The first version of PAW is presented in [1]. A workflow created in PAW is prepared for execution in three steps: First, the tasks are analyzed and the workflow is augmented with associative tasks; the new version of the workflow, which we call the *analyzed* workflow, represents not only the logic flow of the analytics process but also its execution semantics. Second, workflows are manipulated by swapping, composing/decomposing and factorizing/distributing transitions, in order to achieve workflows that have equivalent outputs with their original state, but have a form that can result in optimized execution. Third, PAW schedules the execution of a set of workflows following the novel technique of multi-workflow optimization, on which we focus in this demonstration. This technique is based on the joint execution of the common parts of two or more workflows. PAW can be employed on top of any system that executes analytics processes on big data sources. The platform mediates between users and a set of available data management technologies, such as relational DBMSs, key-value stores and column stores.

2. OVERVIEW OF PAW

PAW is a part of a larger system, called Adaptable Scalable Analytics Platform (ASAP) [6], but it can also stand as an independent tool for workflow management and optimization. Other ASAP components include execution, monitoring, visualization of results, online adaptation, etc. PAW presents a unified interface for users to create, modify, analyze, optimize and execute analytics workflows over a diverse collection

Series ISSN: 2367-2005

10.5441/002/edbt.2017.71

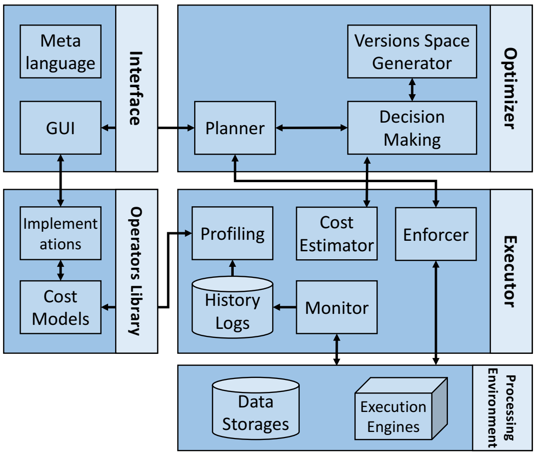


Figure 1: The architecture of PAW

of data stores and processing engines. Figure 1 depicts the architecture of PAW, as well as its interaction with the rest of ASAP. The components of PAW communicate using the internal workflow representation and are:

Operators library. This library contains operators, and their corresponding *implementations* with *cost functions*. The operators are classified as, either logical operators, which perform the core analytics jobs over the data, or the associative operators, which serve as 'glue' between different engines and perform move and transformation operations.

Interface. The *GUI* allows users to interactively create and/or modify a workflow, and add new operators to the *Library*. The user designs a workflow graph in the interactive tool and describes data and operators in the *Tree-metadata language*, which captures structural information, operator properties, and so on.

Optimizer. The orchestration of the optimization process is performed by the *Planner*. It takes as an input a workflow from the *Interface* and sends it to the *Decision Making* module, that returns back an optimized version of a workflow. All possible versions are produced in the *Versions Space Generator* and their costs are estimated by the *Cost Estimator*. The *Decision Making* module chooses the version with the minimal cost as an optimal one.

Executor. The executor performs several tasks. The *Enforcer* schedules workflows for execution, generates executable code and dispatches workflow fragments to execution engines. The *Monitor* observes the system state, tracks the progress of executing workflows and stores *History Logs* of runs. These logs are used to construct more precise *cost functions* of operators through the *Profiling* module. This module in PAW is external, it is also developed as a part of ASAP project, and called IRES [10].

PAW implements a novel workflow model [7, 8]. A workflow W is a directed, acyclic graph (DAG) $G = (V, E)$. The vertices V represent data processing tasks and the edges E represent the flow of data. Each task is a set of *inputs*, *outputs* and an *operator*. Data and operators need to be accompanied by a set of metadata, i.e., properties that describe them. Such properties include input data types and parameters of operators, the location of data objects or operator invocation scripts, data schemas, implementation details, engines etc.

3. MULTI-WORKFLOW OPTIMIZATION

Our technique of multi-workflow optimization (MWO) is based on the joint execution of the common parts of workflows. Specifically, a set of workflows is combined to one joint workflow, so that one or more common subgraphs in these workflows, appear only once in the joint workflow and, therefore, are executed only once. The technique consists of four steps: (1) for each workflow generate all possible equivalent workflow versions and prune them using heuristics; (2) detect common tasks and find the common parts in workflow versions; (3) estimate the processing cost of joint executions; (4) choose workflow versions and common parts in them for the joint execution.

3.1 Generating workflow versions

Two workflow versions are equivalent if they produce the same output, given the same input. We generate all possible versions by applying the following transitions:

Swap. The *swap* transition applies to a pair of vertices, v_1 and v_2, which occur in adjacent positions in a workflow graph G, and produces a new graph G' in which the positions of v_1 and v_2 have been interchanged. The goal of *swap* is to change the execution order of tasks.

Compose. The *compose* transition takes as input two vertices and produces one new vertex that includes the tasks of both initial vertices. The goal of *compose* is to allow for a united optimisation of the tasks included in the two vertices, e.g. joint micro-optimization on an execution engine.

Decompose. The *decompose* transition takes as input one vertex and produces two new vertices that, together, include all the tasks of the initial vertex. The new vertices may or may not be connected. The goal of *decompose* is to lead to separate optimisation of subgroups of the tasks.

Factorize. The *factorize* transition replaces multiple identical vertices that all feed (or are fed by) one *branching* vertex and take as input different datasets, with one such vertex that is performed on the output (input) data of the *branching* vertex. The optimization derives from the fact that the operation of the replaced vertices is performed only once instead of several times, and, moreover, on a reduced in size aggregated dataset.

Distribute. The *distribute* transition replaces one vertex with multiple identical ones, which are distributed on the input (or output) paths of a preceding (or succeeding) *branching* vertex. The optimization opportunity is created either by the parallelization of the execution of the identical vertices, their distribution over the input dataset, or even by the reduction of size of the aggregated input data due to their being pushed toward the root of the workflow.

If the version space is big, exploration methods more efficient than exhaustive search are required. We improve search performance by pruning the space with several heuristics based on the following categorization of operators:

- **Blocking operators** require knowledge of the whole dataset.
- **Non-blocking operators** process each tuple separately.
- **Restrictive operators** output a smaller data volume than the incoming data volume.

R1-2 are a list of rules, following which the process of generating the search space speeds up. Heuristics H1-2 prunes the search space.

- **R**1: Find branching operators and check if they are connected with operators that are identical instances of a logical operator. Try to *factorize* this set of operators.
- **R**2: Find (linear) paths and try to *swap* the operators in

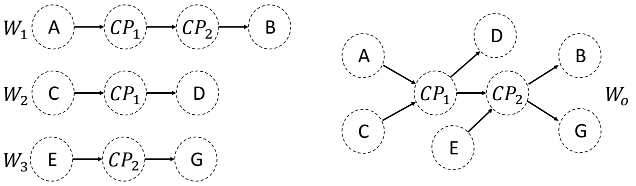


Figure 2: Example of multi-optimization of three workflows

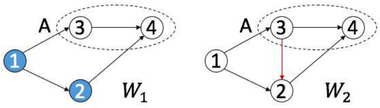


Figure 3: Independently executable and not independently executable subgraphs

each of such paths.

- **H**1: Move restrictive operators to the root of the workflow, e.g. change *extract* → *function* → *filter* to *extract* → *filter* → *function*, if possible.
- **H**2: Group non-blocking operators together and separately from blocking operators, e.g., change *filter* → *sort* → *function* → *group* to *filter* → *function* → *sort* → *group*.

3.2 Creating the joint workflow

A set of workflows $\mathcal{W} = \{W_1, \ldots, W_m\}$ may be combined in a joint workflow denoted as $W_o = W_1 \circ \cdots \circ W_m$. We find common parts in the workflows and use them as joint subgraphs connected with the rest of the workflow graphs. Figure 2 depicts three workflows W_1, W_2, W_3 and a joint workflow of them, W_o. CP_1 and CP_2 represent common parts of W_1, W_2 and W_1, W_3, respectively, and $A..G$ are the remaining parts of workflows.

3.2.1 *Finding common parts*

A common part consists of common tasks. Two common tasks consist of the same operators, inputs and outputs. We detect common tasks by comparing properties of metadata of tasks, such as input and output data schemas, parameters of operators etc.

After detecting common tasks, we look for subgraphs consisting only of common tasks and compare their structures. If such subgraphs are identical, then they constitute a *common part*. Formally, the latter is defined as follows:

Definition 1. A *common part* $CP(W_1, \ldots, W_m)$ of a set of workflows $\{W_1, \ldots, W_m\}$ is a subgraph S, so that S is part of every one of the workflows, i.e. $S \in W_1 \wedge \cdots \wedge S \in W_m$, and operators of corresponding vertices in a subgraph S of every workflow are identical.

3.2.2 *Evaluation of a common part*

After finding a common part, we determine if it can be used for the creation of the joint workflow. We do this based on the concepts of *execution state* and *independently executable subgraph*.

An *execution state* ES of a workflow W is a state for which some of the vertices are assumed to have been executed and no vertices are executing. An *independently executable subgraph* $S \in W$ with respect to some execution state ES_W, is a subgraph that can be executed without executing any vertex in $W \setminus (ES_W \cup S)$.

Figure 3 depicts two workflows W_1 and W_2. In W_1, subgraph A is independently executable with respect to the execution state, the executed vertices of which are colored in blue. In W_2, subgraph A is not independently executable with respect to any execution state, because vertex 4 cannot be executed before vertex 2, and vertex 2 cannot be executed before vertex 3, so vertex 2 has to be executed between vertices 3 and 4.

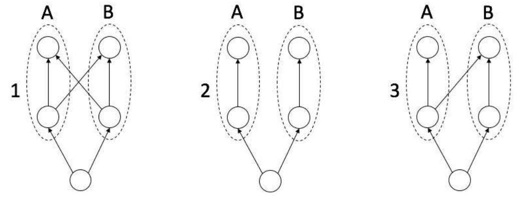


Figure 4: Mutual arrangement of subgraphs A and B

The creation of a joint workflow W_o of a set of workflows $\mathcal{W} = \{W_1, \ldots, W_m\}$ that have one common part CP, is possible if CP is independently executable for some execution state for every $W \in \mathcal{W}$.

3.2.3 *Evaluation of a set of common parts*

A set of workflows to be composed may contain not one, but several common parts. There can be cases for which not all of the common parts can be used for the creation of the joint workflow. To evaluate if a set of common parts $\mathcal{CP}$ can be used in combination for the creation of the joint workflow, we check the *mutual arrangement* of common parts in this set in pairs $CP_i, CP_j \in \mathcal{CP}$.

A *vertex v is reachable* from another vertex u if there is a directed path that starts from u and ends at v. A *subgraph S depends* on vertex v if there exists a vertex u in the subgraph and u reachable from v. The possible *mutual arrangement* of the subgraphs corresponding to two common parts CP_i and CP_j is one of the following (Figure 4):

1. Independent, if there does not exist a pair of vertices $\{v_i, v_j\}$, $v_i \in CP_i, v_j \in CP_j$ for which CP_i depends on v_j or CP_j depends on v_i.
2. CP_i depends on CP_j, if there is a vertex $v \in CP_j$ and CP_i depends on v, but there is not a vertex in CP_i so that CP_j depends on it.
3. CP_i and CP_j are cross-dependent if there are vertices $v_i \in CP_i$, $v_j \in CP_j$ and CP_j depends on v_i and CP_i depends on v_j.

Depending on their mutual arrangement in the set of workflows, a pair of common parts can be selected for the construction of the joint workflow or not: If the common parts are mutually arranged as (1) in all workflows, both can be selected; if they are mutually arranged as in (3), even in one workflow, they cannot be both selected. If they are mutually arranged as in (2) in some of the workflows, they can be both selected if they have the same dependency in all these workflows. Hence, in some cases, we are forced to select only some of the common parts. We do this based on the estimation of processing cost of different choices for the construction of the joint workflow.

3.3 Estimation of processing costs

We estimate the performance and cost of operators by actually running the operator in representative configuration combinations. Using these measurements, surrogate estimator models are trained that can be used to approximate operators performance for non-tested configurations. The processing cost of a workflow W, C_W, is the sum of the cost of the its tasks: $C_W = \sum_{i=1}^{n} C_{T_i}$.

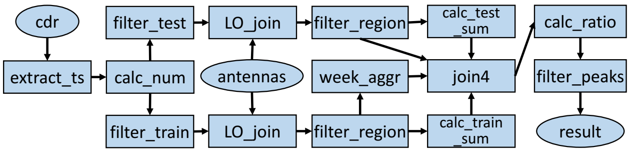


Figure 5: ‘Peak Detection’ workflow

Let us consider a pair of workflows $\{W_1, W_2\}$ with a common part CP and execution states ES_1 and ES_2, respectively. The cost of the joint workflow $W_o = W_1 \circ W_2$ is the sum of the cost of execution states $C(ES_1)$ and $C(ES_2)$, the cost of the common part $C(CP)$, the costs of the rest of workflows $C(W_1 \setminus \{CP, ES_1\})$, $C(W_2 \setminus \{CP, ES_2\})$, and a synchronization cost $C(sync)$, which captures the cost for creating the joint workflow:

$$\begin{aligned} C(W_1 \circ W_2) = C(ES_1) + C(ES_2) + C(CP) + C(sync)+ \\ +C(W_1 \setminus \{CP, ES_1\}) + C(W_2 \setminus \{CP, ES_2\}) = \\ = C(W_1) + C(W_2) - C(CP) + C(sync) \end{aligned}$$

The processing cost of workflows $\mathcal{W} = \{W_1, \ldots, W_m\}$ with common parts $\{CP_1, \ldots, CP_n\}$ is:

$$\begin{aligned} C(W_1 \circ \cdots \circ W_m) = \\ = \sum_{i=1}^{m} C(W_i) - \sum_{i=1}^{n}((n_i - 1)C(CP_i) - C(sync_i)) \end{aligned}$$

where n_i is the number of occurrences of common part CP_i in $\mathcal{W}$. After estimating the processing costs of all workflow versions and common parts, exhaustive search chooses common parts and workflows with the lowest cost.

3.4 Online Multi-Workflow Optimization

MWO is applicable, when the user launches multiple workflows simultaneously. Usually, a frequent case is when PAW receives new workflows one by one or in batches, while some workflows are currently executing. To cover this case PAW offers an Online Multi-Workflow Optimization (OMWO). It re-optimizes currently running workflows on each addition of a new workflow to our platform. As soon as a new workflow is inserted to PAW the optimizer gets the current states of execution of workflows, i.e. which vertices have been executed, are executing and have not yet started execution. Next, it applies MWO to a set of workflows, that consist of the new workflow and not-executed parts of workflows that are currently executing. Their intermediate results are used as inputs in these partial workflows.

4. DEMONSTRATION

In the following, we describe the proposed demonstration.

System setup. PAW is demonstrated on a cluster, with the following configuration: The cluster consists of 4 server-grade physical nodes. Each one of those is equipped with a 3rd generation i5 CPU (@ 2.90 GHz) and 16GB of physical memory and an array of two HDDs on RAID-0. The operating system is Debian 6 (squeeze) Linux. For the time being, three software platforms are running: Hadoop (CDH 4.6.0), Spark (1.4.1) and Weka (3.6.13).

Workloads. The demonstration uses synthetic and real workflows on real data. The synthetic workflows are constructed based on ETL benchmarking [9]. Real workflows and data come from the two use cases of ASAP [6] and belong to the domains of telecommunications and web analytics. Figure 5 displays one of the telecommunication workflows. The telecommunication use case involves processing anonymised Call Detail Records (CDR) data collected in Rome for 2015 year and stored in HDFS. All workflows' operators have implementations in Spark and Postgres. The web analytics use case involves anonymization of web content (WARC files) stored in ElasticSearch. The workflows are implemented in Spark and run over varying data set sizes ranging from 1 million to 4 billion rows. There are two types of workflows: one models entity recognition/disambiguation and k-means, and another models continuous processing of incoming data, e.g., subscription/notification at scale.

Demonstration scenarios. The demonstration focuses on the multi-workflow optimization functionality of PAW. It includes three types of scenarios that aim to show a distinct view of the benefit of our novel technique and create discussion on the potential of multi-workflow optimization. The demonstration is interactive with the audience. The participants are invited to experience all functionalities of PAW, create workflows from scratch or change existing ones, watch the automated management of the workflow as well as review the internals of the platform, e.g. internal workflow representation. They are also enabled to play with the management of multiple workflows, by selecting workflows for optimization and execution, pausing and resuming execution, selecting common parts for optimization, etc.

Scenarios A. Their goal is the comparison of single and multi-workflow optimization. We show exemplary cases of small sets of workflows, in which the versions selected by single-workflow optimization differ or identify with the versions selected by multi-workflow optimization.

Scenarios B. Their goal is to show the overall performance of multi-workflow optimization for a variety of workflow workloads. The workloads include workflows with a variety of tasks, short-running and long-running, in a variety of combinations, with an emphasis on long chain paths or numerous parallel paths. Also, the scenarios show how the existence of common parts affects optimization, by varying their number and their size.

Scenarios C. Their goal is to show the continuous arriving, optimization and execution of workflows. We create time series of workflows from scenarios B. We emphasize in the effect of ranging the size of the window on the arrival timeline, within which workflows are optimized by online multi-workflow optimization.

5. REFERENCES

[1] M. Filatov and V. Kantere. PAW: A Platform for Analytics Workflows. In *EDBT*, 2016.

[2] Oracle warehouse builder 10g. http://www.oracle.com/technology/products/warehouse/.

[3] Informatica ‘powercenter’. http://www.informatica.com/products/powercenter/.

[4] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Optimizing analytic data flows for multiple execution engines. In *ACM SIGMOD*, 2012.

[5] A. Simitsis, K. Wilkinson, U. Dayal, and M. Hsu. HFMS: Managing the lifecycle and complexity of hybrid analytic data flows. In *ICDE*, 2013.

[6] Asap. http://www.asap-fp7.eu/.

[7] V. Kantere and M. Filatov. A framework for big data analytics. In *C3S2E*, 2015.

[8] V. Kantere and M. Filatov. Modelling processes of big data analytics. In *WISE*, 2015.

[9] A. Simitsis, P. Vassiliadis, U. Dayal and V. Tziovara. Benchmarking ETL workflows. In TPCTC, 2009.

[10] K. Doka, N. Papailiou, D. Tsoumakos and N. Koziris: IReS: Intelligent, Multi-Engine Resource Scheduler for Big Data Analytics Workflows. In SIGMOD, 2015.

Efficient spatio-temporal event processing with STARK

Stefan Hagedorn
TU Ilmenau, Germany
stefan.hagedorn@tu-ilmenau.de

Timo Räth
TU Ilmenau, Germany
timo.raeth@tu-ilmenau.de

ABSTRACT

For Big Data processing, Apache Spark has been widely accepted. However, when dealing with events or any other spatio-temporal data sets, Spark becomes very inefficient as it does not include any spatial or temporal data types and operators. In this paper we demonstrate our STARK project that adds the required data types and operators, such as spatio-temporal filter and join with various predicates to Spark. Additionally, it includes k nearest neighbor search and a density based clustering operator for data analysis tasks as well as spatial partitioning and indexing techniques for efficient processing. During the demo, programs can be created on real world event data sets using STARK's Scala API or our Pig Latin derivative *Piglet* in a web front end which also visualizes the results.

1. INTRODUCTION

Spatio-temporal data is used in various application areas: for example by (mobile) location aware devices that periodically report their position as well as in news articles describing *events* that happen at some time and location. Spatio-temporal event data can, e.g., be extracted from text documents using spatial and temporal taggers that identify the respective expressions in a text corpus. The extraction of the structured event data from text is just a first step and data needs to further be analyzed using appropriate data mining operations to gain new insight.

As the event data sets may become very large, scalable tools are needed for the event analysis pipelines. Apache Spark has become a very popular platform for such Big Data analytics because of its in memory data model that allows much faster execution than with Hadoop MapReduce programs. However, Spark has a general data model which does not take the spatial and temporal aspects of the data into account, e.g., for partitioning. Furthermore, dedicated data types and operators for this spatio-temporal are missing.

In this paper we demonstrate our STARK[1] framework for scalable spatio-temporal data analytics on Spark, with the following features:

- STARK is built on top of Spark and provides a domain specific language (DSL) that seamlessly integrates into any (Scala) Spark program.
- It includes an expressive set of spatio-temporal operators for filter, join with various predicates as well as k nearest neighbor search.
- A density based clustering operator allows to find groups of similar events.
- Spatial partitioning and indexing techniques for fast and efficient execution of the data analysis tasks.

In contrast to similar existing solutions for Spark, STARK is the only framework that addresses not only spatial but also spatio-temporal data. Unlike other frameworks, STARK is seamlessly integrated into the Spark API so that spatio-temporal operators can directly be called on standard RDDs. Furthermore, we provide a Pig Latin extension in our Piglet engine to create (spatio-temporal) data processing pipelines using an easy to learn scripting language. A web front end supports users with interactive graphical selection tools and also visualizes the results. We evaluated STARK in a mirco benchmark against other solutions and showed that we can outperform them.

2. THE STARK FRAMEWORK

STARK is tightly integrated into the Apache Spark API and users can directly invoke the spatio-temporal operators and their RDDs. To achieve this, we created new data type and operator classes that make use of already existing Spark operations, but also extend internal Spark classes. Figure 1 gives an overview of STARK's architecture and its integration into Spark.

In the following, we describe the internal components for spatial partitioning and indexing as well as the API/DSL for spatio-temporal operations and integration into Spark.

2.1 Partitioning

Partitioning has a significant impact in data parallel platforms like Spark. If the partitions sizes, i.e., the number of elements per partition, are not balanced, a single worker node has to perform all the work while other nodes idle.

Spark already includes partitioners, but they do not exploit the spatial (or spatio-temporal) characteristics. Spatial-temporal partitioning means that partitions are not created by using, e.g., a simple hash function, but by considering the location in space and/or time of occurrence. Thus, after

[1] https://github.com/dbis-ilm/stark

Series ISSN: 2367-2005

10.5441/002/edbt.2017.72

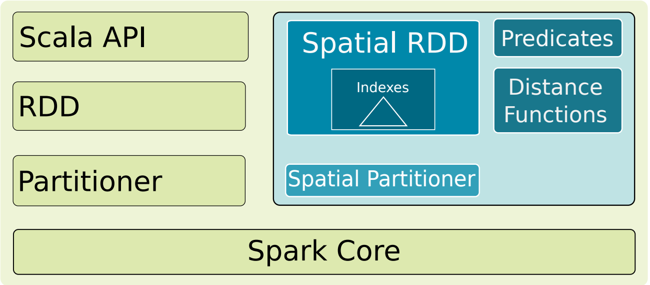


Figure 1: Overview of STARK architecture and integration into Spark.

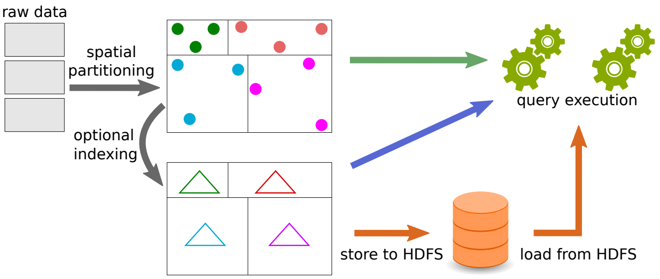


Figure 2: Internal workflow for converting, partitioning, and querying spatio-temporal data

a spatio-temporal partitioner was applied on a data set, a partition contains all elements that are near to each other in time and/or space and the bounds of a partition represent a spatial region and/or temporal interval which cover all items of that partition. This bound is very useful to determine what partitions actually have to be processed for a query. For example, an *intersects* query only has to check the items of partitions where the partition bounds themselves intersect with the query object. Such a check can decrease the number of data items to process significantly and thus, also reduce the processing time drastically.

When the spatial and temporal objects of a data set are not points or instants, respectively, these regions and intervals may span across multiple partitions. There are two options to handle such scenarios:

- The item is replicated into every of these partitions and the resulting duplicates have to be pruned afterwards.
- The items are assigned to only one partition and the partition bounds are adjusted accordingly which results in overlapping partitions.

STARK uses the latter approach by assigning polygons to partitions based on their centroid point. Beside the partition bounds, we keep an additional *extent* information that is adjusted with the minimum and maximum values of the respective objects in each dimension. We decide which partition has to be checked during query execution based on this *extent* information and prune partitions that cannot contribute to the final result.

In its current version, STARK only considers the spatial component for partitioning. The partitioners implement Spark's `Partitioner` interface and can be used to spatially partition an RDD with the RDD's *partitionBy* method.

Grid Partitioner.

The first partitioner included in STARK is a fixed grid partitioner. Here, the data space is divided into a number of intervals per dimension resulting in a grid of rectangular cells (partitions) with equal dimensions. The bounds of these partitions are computed in a first step and afterwards with a single pass over the data, each item is assigned to a partition by calculating in which grid cell this item is contained.

Cost-Based Binary Space Partitioner.

As the fixed grid partitioner created partitions of equal size over the data space, it might create some partitions that contain the majority of the data items, while other partitions are empty. As an example consider the world map where events only occur on land, but not on sea. With a grid partitioning, there might be empty cells on sea and overfilled partitions in densely populated areas. To overcome this problem, we implemented a cost based binary space partitioning algorithm, based on [1]. This partitioner divides the space into two partitions with equal cost (number of contained items). If the cost for one partition exceeds a threshold, it is recursively divided again into two partitions of equal cost. This way, large regions with only a few items will belong to the same partition, while dense regions are split into multiple partitions. The recursion stops when a partition does not exceed the cost threshold or the algorithm reached a granularity threshold, i.e., a minimum side length of a partition.

2.2 Indexing

Just as in relational DBMS, indexing the content can significantly improve query performance. STARK uses the JTS[2] library for spatial operations. This library also provides an R-tree implementation (more accurately, an STR-tree) for indexing. STARK can use this index structure to index the content of a partition. A spatial partitioning is not mandatory to use index, but might bring additional performance benefits. Basically, STARK has three indexing modes, that can be chosen by the user:

No Indexing.

The partitions are not indexed and all items within a partition have to be evaluated with the respective predicate function.

Live Indexing.

When a partition is processed for evaluating a predicate, the content of that partition is first put into an R-tree and then, this index is queried using the query object. Since the results of the R-tree query are only candidates where the minimum bounding boxes match the query, these candidates have to be checked again if they really match the query object. During this candidate pruning step, the temporal predicate is evaluated as well, if needed. Live indexing can be used in a program by calling the *liveIndex* method on an RDD. This method takes the order of the tree as well as an optional partitioner as parameters, in case the RDD should be repartitioned before indexing.

Persistent Indexing.

Creating an index may be time consuming and often the same index will be reused in subsequent runs of the same or in another program. For such cases, STARK allows to

[2] http://tsusiatsoftware.net/jts/main.html

persist the index to disk/HDFS using Spark's method to save binary objects. An indexing that should be persisted can also be used by that same program. Thus, users don't need to do an extra run to just persist the index, but can already perform their operations. Such an index mode is done using the *index* method, which also takes the order of the tree as well as an optional partitioner as parameter.

2.3 DSL

One important design goal of STARK was to create an DSL that can be intuitively used by users within any (Scala) Spark program. This DSL provides all required operations for flexibly working with spatio-temporal data. This means that raw data loaded from HDFS or any other source can easily be processed by spatio-temporal operators and may be spatially partitioned and optionally indexed. The partitioning and indexing is transparent to the subsequent query operators which means they can be executed with or without spatial partitioning and indexing (and any combination thereof). Furthermore, the created indexes can be materialized, e.g., to HDFS, and be re-used within other programs. Figure 2 gives an overview of these possibilities.

In order to represent spatio-temporal data, STARK provides the `STObject` class. This class has only two fields: (1) `geo` that stores the spatial attribute and (2) and optional `time` field which holds the temporal information of an object. The `time` is optional to support spatial-only data that does not need any temporal information.

Beside these fields, the `STObject` class provides methods which check the relation to other spatio-temporal objects:

intersect(o) checks if the two instances (*this* and `o`) intersect in their spatial and/or temporal component,

contains(o) tests if *this* object completely contains `o` in their spatial and/or temporal component, and

containedBy(o) which is implemented as the reverse operation of *contains*

A formal definition for two objects o and p of type `STObject` and a predicate Φ can be given as:

$$\Phi(o,p) \Leftrightarrow \Phi_s(s(o),s(p)) \wedge (\quad (1)$$

$$(t(o) = \perp \wedge t(p) = \perp) \vee \quad (2)$$

$$(t(o) \neq \perp \wedge t(p) \neq \perp \wedge \Phi_t(t(o),t(p)))) \quad (3)$$

Where $s(x)$ denotes the spatial component of x, $t(x)$ the temporal component of x, Φ_s and Φ_t denote predicates that check spatial or temporal objects, respectively, and $\perp$ stands for `undefined` or `null`. This says that the predicate Φ is true for two spatio-temporal objects o and p, if the predicate on the spatial components of o and p is true (1), and both temporal components are *not* defined (2), or they are defined and the predicate on the temporal components of o and p is true as well (3).

To add the spatio-temporal operations to an RDD, STARK implements a special helper class called `SpatialRDDFunction` that has one plain Spark RDD as attribute and implements the supported spatio-temporal operations. In plain Spark, when an RDD contains 2-tuples of `(k,v)` an implicit conversion method creates a `PairRDDFunction` object, which provides, e.g., the *join* functionality using `k` as the join key. STARK follows the same approach: for an RDD of 2-tuples `(k,v)` we create a `SpatialRDDFunction` object implicitly, if `k` is of type `STObject`[3]. This implicit conversion is transparent to users and creates a seamless integration into any Spark program. Users don't have to explicitly create an instance of any of STARK's classes (except `STObject`) to use the spatio-temporal operators.

STARK has an *intersects*, *contains*, and *containedBy* predicate. In addition to that, we support a *withinDistance* operation, which finds all elements that are within a given maximum distance around the query object. Here, the distance function can be passed as a parameter so that users can implement their own function and adjust STARK to their requirements. However, we also include standard distance functions that can be used out of the box. Furthermore, there is a k nearest neighbor search operator.

An important data mining operation is clustering. STARK implements the DBSCAN algorithm for Spark inspired by MR-DBSCAN for MapReduce described in [1]. The implementation exploits the spatial partitioning: points that are within ϵ-distance from the partition border (where ϵ is the DBSCAN parameter), are replicated into the respective neighboring partition. In a next step a local partitioning is performed locally and in parallel on each partition. In a subsequent merge step, these local clusterings are merged using the replicated points, which may connect two clusters to a single one.

The following example shows the usage the spatio-temporal operator on an RDD with STARK. Consider an input file with a schema *(id: Int, category: String, time: Long, wkt: String)*. After pre-processing, we get an RDD of exactly that type: `RDD[(Int, String, Long, String)]`. We then create an `STObject` representing the location from the WKT string and time of occurrence from the `time` field of each entry:

```
val events = rawInput.map {
 case (id, ctgry, time, wkt) =>
          ( STObject(wkt, time), (id, ctgry) ) }
```

The `events` RDD of type `RDD[(STObject, (Int, String))]` and can now be used with any supported spatial-temporal predicate function:

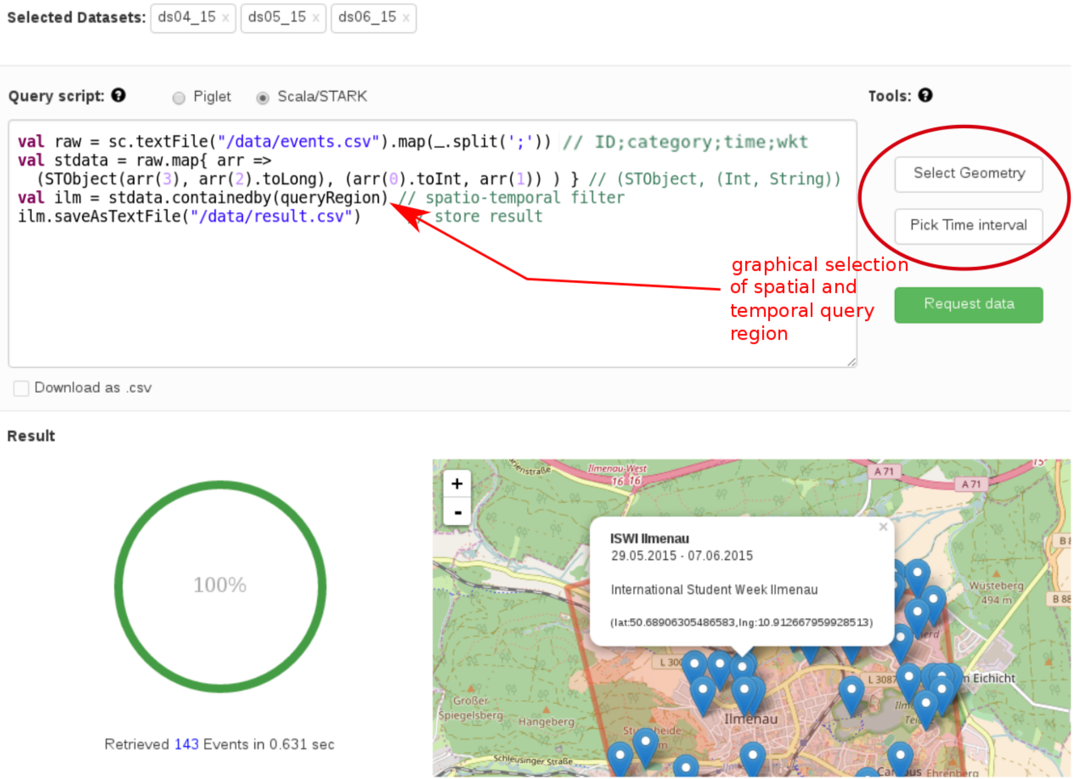
```
val qry = STObject("POLYGON((...))", begin, end)
val contain = events.containedBy(qry)
val intersect = events.liveIndex(order = 5)
                          .intersect(qry)
```

We create a query object with a spatial polygon defined as a WKT string and a temporal interval. Here `begin` and `end` are `Long` values that describe the begin and end of a temporal time window for querying. With the *containedBy* function we can find all items in the events RDD that are contained by the query object. In the second example, the RDD is indexed using live indexing with an order of the R-tree of 5. We can then simply call the *intersects* (or any other supported function) on that indexed RDD.

3. EVALUATION

We evaluated our STARK implementation against other existing Hadoop- and Spark-based solutions for spatial data processing. In this evaluation we looked at provided features and further performed a micro benchmark. During this evaluation we found that not only do some systems have serious bugs and produce wrong results, but they are also not intuitively to use and have a very limited or even no API (only a

[3] i.e., `RDD[(STObject , V)`, where V can be any type.



```
val raw = sc.textFile("/data/events.csv").map(_.split(';')) // ID;category;time;wkt
val stdata = raw.map{ arr =>
  (STObject(arr(3), arr(2).toLong), (arr(0).toInt, arr(1)) ) } // (STObject, (Int, String))
val ilm = stdata.containedby(queryRegion) // spatio-temporal filter
ilm.saveAsTextFile("/data/result.csv")    // store result
```

Retrieved 143 Events in 0.631 sec

ISWI Ilmenau
29.05.2015 - 07.06.2015

International Student Week Ilmenau

(lat:50.68906305486583,lng:10.912667959928513)

Figure 3: The user interface for querying data from the repository.

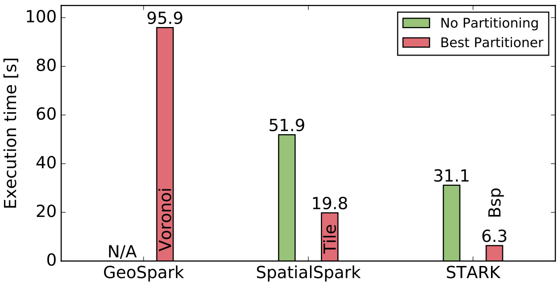


Figure 4: Execution times for self join operation for best partitioner and indexing.

command line interface). Figure 4 shows the result of a self join operation on a data set with 1,000,000 points comparing STARK with the Spark-based frameworks SpatialSpark [2] and GeoSpark [3]. The figure shows the execution time without partitioning as well as for the partitioner that resulted in the fasted execution time. For GeoSpark we experienced different result counts in each repetition of the experiment for two spatial partitioners. The results show that STARK outperforms the other frameworks in both cases. More results of the performance evaluation can be found in our GitHub repository[4].

4. DEMONSTRATION SCENARIOS

STARK is integrated into a larger project in which event information is extracted from text articles, stored as structured data, and analyzed using STARK's operators. We will prepare real world data sets with events from Wikipedia (created in the context of that project) as well as other spatio-temporal data sets with different contents. During the demonstration, visitors will be able to create and execute simple queries and complex data analysis pipelines or choose from prepared programs using our web front end. For that we will prepare different real world use case queries that include (reverse) geocoding, spatio-temporal join and aggregation, as well as clustering/co-location. Figure 3 shows the web front end with the query interface which supports the formulation of the spatio-temporal components by providing graphical selection tools using maps and date/time pickers that make the selected values available in the program. Queries and pipelines can be created as Scala programs, but we also allow to create these programs as Pig Latin scripts using our Piglet [4] engine that extends the original Pig Latin language with the before mentioned data types and operators. The results of the queries will be dynamically visualized in the web front end.

Acknowledgments.

This work was partially funded by the German Research Foundation (DFG) under grant no. SA782/22.

5. REFERENCES

[1] Y. He, H. Tan *et al.*, "MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data," *FCS*, vol. 8, no. 1, pp. 83–99, 2014.

[2] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud." ICDEW, 2015.

[3] J. Yu, J. Wu, and M. Sarwat, "Geospark: A cluster computing framework for processing large-scale spatial data." SIGSPATIAL, 2015, p. 70.

[4] S. Hagedorn and K.-U. Sattler, "Piglet: Interactive and platform transparent analytics for rdf & dynamic data," in *WWW*, April 2016, pp. 187–190.

[4] https://github.com/dbis-ilm/spatialbm

Context-Aware Proactive Personalization of Linear Audio Content

Paolo Casagranda
Rai Radiotelevisione italiana
and University of Torino
Torino, Italy
paolo.casagranda@rai.it

Maria Luisa Sapino
University of Torino
Torino, Italy
mlsapino@di.unito.it

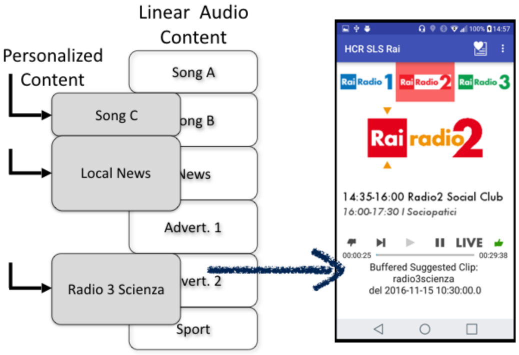
K. Selçuk Candan
Arizona State University
Tempe, USA
candan@asu.edu

ABSTRACT

How many times did you wish the radio programming was more aligned with your interests or current situation? How many times did you feel the need to change the channel because of a non-interesting content on your favorite station? Did you ever feel distracted by the audio programming in your car at a busy intersection? We present a platform for *proactive personalization of linear audio content* within a hybrid content radio framework. Hybrid content radio programming aims at enhancing the traditional broadcast radio experience and augmenting it with audio content related to the listener's context. It allows enrichment of the broadcaster's program schedule with context-aware, personalized audio content, with the goal of improving the users' listening experience, decreasing their propensity to channel-surf, and giving them more targeted content, such as local news, entertainment, music and also relevant advertisements. Differently from most of the popular commercial recommendation-based streaming music services, hybrid content radio systematically and automatically adds audio content to an existing, linear audio structure. More specifically, part of the linear content is *replaced*, in a proactive way, with *content relevant to the user's current context* – i.e., profile, emotional state, activity, geographical position, weather, or other factors contributing to the state of the listener. In addition to enabling functional enhancement of the radio experience, the presented framework also supports network resource optimization, allowing effective use of the broadcast channel and the Internet.

CCS Concepts

•**Information systems → Recommender systems; Location based services; Multimedia streaming; Speech / audio search;**



14:35-16:00 Radio2 Social Club
16:00-17:30 I Sociopatici

Buffered Suggested Clip:
radio3scienza
del 2016-11-15 10:30:00.0

Figure 1: The hybrid content radio audio replacement concept, and a view of the prototype app playing the clip.

Keywords

Personalization; Recommender Systems; Context-based recommendations; Location-based services; Radio.

1. INTRODUCTION

We present a novel platform for *context-aware proactive personalization of linear audio content* within a hybrid content radio framework, so we specifically focus on audio content delivery. The Proactive Personalyzed Hybrid Content Radio (or PPHCR) system we present is part of the evolution of traditional linear radio, as described at the International Broadcasting Convention in 2015 [6] and in [7, 10]. PPHCR is based on the live radio streams and associated metadata from Rai, the Italian Public Service Media Company. Beside dealing with all the intermediate scenarios where the broadcaster provides a linear programming, hybrid content radio recommends enriching and context-based online content.

1.1 Context-Driven Content-Delivery in Hybrid Content Radio

Hybrid content radio aims at making linear broadcast radio more flexible, allowing seamless replacement of parts of the broadcast audio content with relevant audio content mined from recent podcasts and content archives. The basic metadata descriptions enabling this service come from the ETSI Standards created by the RadioDNS Project, see [9]. Figure 1 illustrates the content replacement concept, with

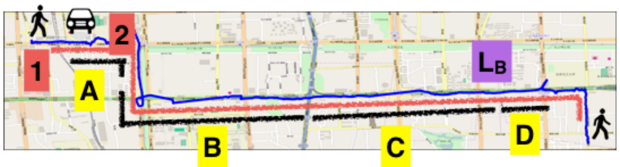


Figure 2: Audio content is recommended following contextual information, such as listener's position and route. When the user's car starts moving, the system predicts a travel duration ΔT, and tries to allocate the most relevant content for the available time ΔT, recommending media items A, B, C, D. Item B is also relevant to location L_B the user will reach.

the user interface of the prototype client app.

Context-aware recommender systems have been subject to increasing attention in the last few years, see [2]. Some studies specifically focus on location awareness, specially those related to the mobile context, see for example [14] and [11].

The proposed PPHCR prototype takes advantage of a novel **proactive recommender system** (PRS), see [5, 13], capable of deciding the time of the recommendation delivery, as described in Section 1.2. The recommender system provides contextually-relevant alternative audio content that will replace part of the broadcast content thus increasing the user's satisfaction and decreasing her tendency to switch channels. Using linear radio as the basic building block for personalized radio has two main advantages:

- the relevance of the content for the listeners increases, enriching their experience, while they keep on listening their favorite radio station
- the efficiency of content delivery can be optimized, if the device allows using a broadcast technology to receive the audio from the broadcast channel

The core novelty of the proposed service, compared to other existing approaches, consists in the joint usage of linear radio personalization and in the type of proactive context-based recommendations, based on the listener's location, movement and preferences. PPHCR creates personalized audio content suggestions accounting for uncertainties in the mobile user's future path as well as driving conditions during the scheduling and delivery of a highly relevant and enjoyable, and yet non-distracting hybrid-audio content.

Figure 2 illustrates an example of proactive recommendation for a driver: the system predicts the route and travel duration and maximizes the relevance of the recommended media items, based on a combination of learned user preferences and geographic relevance of the content. As illustrated in this example, effective delivery of personalized linear radio, with context-based content recommendations, requires to solve the problem of integrating linear schedule timings, spatial information and listeners' preferences, making the solution innovative with respect to existing proposals [4].

1.2 System Description

The functionality of the system relies on the server architecture shown in Figure 3 and on the client PPHCR app that provides the user interface and collects data relevant to discover the context. The content server is the integration of several components cooperating to offer a real-time, personalized radio service. Radio Rai, the radio division of Rai Public Service Media Company, directly provides 10 live 96kbps audio streams [1], the editorial version of more than 100 podcasts created every day and the associated schedule metadata are used to populate the **content repository** and the **metadata DB**. The podcasts are classified by the **clip data management** component according to their categories. News programs, including large parts of speech, are analyzed using an automatic speech recognizer trained with the Italian language. The extracted text is then classified with a Bayesian classifier trained with a set of news, according to a set of 30 categories spacing from art to culture, music, economics.

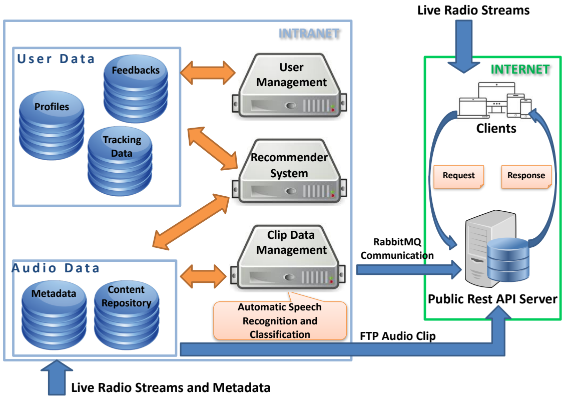


Figure 3: Simplified architecture of the context-aware personalized radio system

User data are organized by the **user management** component. The user's demographic details are stored in the **profiles DB**. The **feedbacks DB** hosts content navigation logs sent by the listener's app together with the implicit or explicit rating given by the user. The **tracking data DB** is a PostGIS based spatial DB with the listener's geographical information. The amount of GPS data arriving to the tracking data DB requires to periodically process and simplify them, extracting a compact, discrete model which describes destination, trajectory, speed, frequency, time of the day and *complexity*. Major *staying points* on the driving paths are calculated using a density based location clustering [8] and complexity is calculated analysing the trajectory simplified using the Ramer-Douglas-Peucker algorithm (RDP).

Contextually relevant linear content recommendations are provided by the **recommender system** component using both user and audio data clusters. For each user the recommender filters a candidate set of media items using content-based relevance based on past listener's feedbacks. Then a compound relevance score is calculated through weighted combination of the content-based relevance and the context-based relevance (location, trajectory, speed and time information). The recommender system then uses the this score to identify the recommendation set of content to be delivered to the listener according to a relevance objective function and temporal scheduling and presentation constraints, taking into account driving conditions as well as driver's projected distraction levels at intersections and roundabouts at user's projected driving path.

1.3 Client Android App

The client PPHCR app, whose interface is shown on the right side of Figure 1, has been implemented on Android mobile OS. The listener can choose one of the live radio services, change service, pause, or skip content. While the user is listening to the service, a positive implicit feedback is periodically sent for that audio content. In contrast, each skip action generates a negative feedback. The app synchronizes metadata and implements buffering and synchronization to ensure that the selected live audio is seamlessly replaced by the recommended clips.

The controls are shown in Figure 1: the user can manually give a feedback, skip current program or navigate the favorite media items.

2. DEMONSTRATION OUTLINE

The demonstration is centered on the personalization of live radio for a listener on the move, in her car. The movement can be real or simulated using an Android third party app providing fake locations (see for example [12]). The personalization works in a proactive way, using a proactive recommender system as described in [13, 3, 5]. The PPHCR App collects and sends to the user management module user's preference data and the GPS locations of the moving listener, allowing to predict the trajectory she's following. Using the listener's feedback and the routes collected when she moves, the system learns to suggest and play content independently from an explicit user action. The key tasks enabled by the prototype are the following:

implicit and explicit user feedback the user can give implicit feedback to the content skipping it, or explicit feedback with the like/dislike buttons

manual skip users can skip live programs and, thanks to buffering synchronized with program schedule metadata, seamlessly replace them with recommended content

proactive recommendations destination, trajectory, speed and available time predictions allow to proactively suggest a list of media items; geographic information is also used to refine recommendations

editorial recommendations injection the editor can selectively choose and inject recommended audio content to specific users

2.1 Demonstration Scenarios

We present two demonstration scenarios: in the first one, the content change is manually triggered by the listener; in the second, the enrichment of the audio content is triggered by a real-time change in the listener's context – more specifically, listener's movement in space.

2.1.1 *Manual Program Change*

While listening to linear radio, from a broadcast channel such as analog FM or digital DAB+, or from the Internet, the user can sometimes wish to listen different content. Greg is passionate about technology and economy, often listening to programs on this topic during the day. This morning there is an endless discussion about football results on his favorite radio channel. He is about to zap channel, but now his radio app allows him to *skip* the live program and surf a list of suggested audio clips. After two skips Greg reaches one of his favorite programs: "Wikiradio".

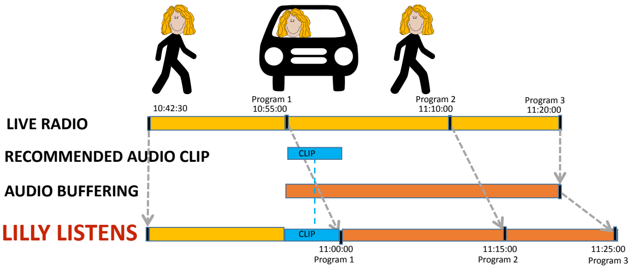


Figure 4: The app recommends an audio clip while Lilly is driving to work (timeline).

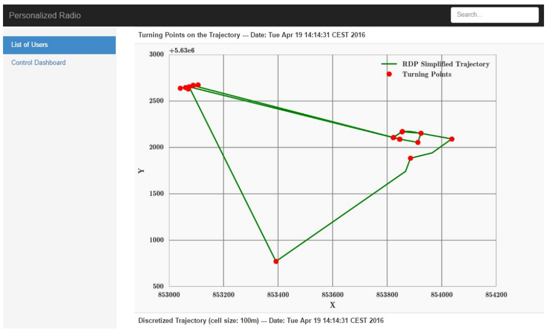
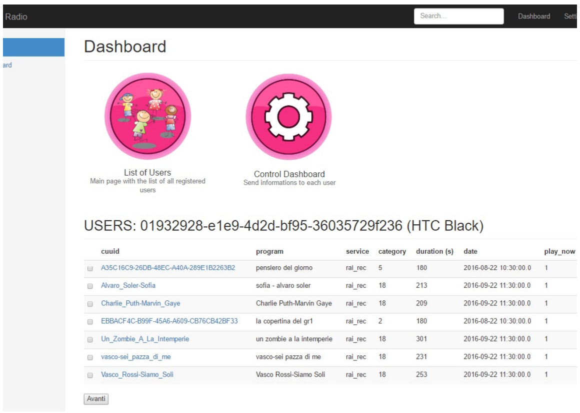

Personalized Radio
Search...
List of Users
Control Dashboard
Turning Points on the Trajectory --- Date: Tue Apr 19 14:14:31 CEST 2016
+5.63e6
3000
2500
2000
1500
1000
500
Y
853000
853200
853400
853600
853800
854000
854200
X
RDP Simplified Trajectory
Turning Points
Discretized Trajectory (cell size: 100m) --- Date: Tue Apr 19 14:14:31 CEST 2016

Figure 5: Control Dashboard: map with the last listener's movements

2.1.2 *Contextual Proactive Recommendation*

Lilly is a young researcher and an appreciated amateur chef. She always tries new recipes and lets her colleagues taste the results. Lilly also likes listening radio while she drives from home to work in the morning and from work to home in the evening. Sometimes music, more often radio talks and discussions on several subjects, especially to those related to food, recipes and cookery. In the past it was difficult to find an interesting program. She tried to record audio programs in the evening for the morning, but it was tricky. This morning is different: her radio app had an update some weeks ago. After she has been driving for some minutes, the PPHCR App *automatically plays* the last news and, after this, an audio clip from "Decanter" program, discussing the differences between French Champagne, Spanish Cava and Italian Prosecco. After that she is pleased to hear the jokes of the time shifted live "The rabbit's roar": the program began 20 minutes ago, but the app can still smoothly present it after "Decanter". She listens, pleased and interested, forgetting the skip button. Figure 4 shows the time-line of the personalization process for Lilly. The suggestions are based on her previous skip history and the knowledge of the context.

2.2 Control Dashboard


Radio
Search...
Dashboard
Sett
ard
Dashboard
List of Users
Main page with the list of all registered users
Control Dashboard
Send informations to each user
Avanti

USERS: 01932928-e1e9-4d2d-bf95-36035729f236 (HTC Black)

	cuuid	program	service	category	duration (s)	date	play_now
□	A35C16C9-26DB-48EC-A40A-289E1B2263B2	pensiero del giorno	rai_rec	5	180	2016-08-22 10:30:00.0	1
□	Alvaro_Soler-Sofia	sofia - alvaro soler	rai_rec	18	213	2016-09-22 11:30:00.0	1
□	Charlie_Puth-Marvin_Gaye	Charlie Puth-Marvin Gaye	rai_rec	18	209	2016-09-22 11:30:00.0	1
□	EBBACF4C-B99F-45A6-A609-CB76CB42BF33	la copertina del gr1	rai_rec	2	180	2016-08-22 10:30:00.0	1
□	Un_Zombie_A_La_Intemperie	un zombie a la intemperie	rai_rec	18	301	2016-09-22 11:30:00.0	1
□	vasco-sei_pazza_di_me	vasco-sei pazza di me	rai_rec	18	231	2016-09-22 11:30:00.0	1
□	Vasco_Rossi-Siamo_Soli	Vasco Rossi-Siamo Soli	rai_rec	18	253	2016-09-22 11:30:00.0	1

Figure 6: Control Dashboard: list of recommendations to send to a specified user

During the demonstration a web-based control dashboard will be used to visualize the users' behavior during the experimentation. The website visualizes the user's past trajectories, content preference, and the details of the recommendation process (see Figure 5). The dashboard also allows to manual injection of recommendations help test recommendations (Figure 6).

3. CONCLUSIONS AND FUTURE WORK

We presented a Proactive Personalized Hybrid Radio system, capable of enhancing the linear radio stream, proactively proposing targeted audio content. The recommended audio items list and the time to show it is generated using the listener's past preferences and the prediction of her current movement. The key contributions are the location and movement information awareness to decide both the time and items to recommend, and the integration of live broadcast and personalized audio content. The system allows a listener-centric radio experience, with the possibility of explicit content skips and proactive audio content recommendations, while preserving the appeal and sense of connection between listeners of traditional broadcast radio. During the demonstration, the audience will be able to observe both the user experience (through the client mobile app) as well as the data-flow and recommendation generation process (through the web based control dashboard).

For the future, we are planning to estimate the geographic relevance of audio items available in the archives. This operation involves the analysis of informative and entertainment content as well as advertisements, validated by a user trial. Furthermore, we plan to create recommendations list taking into account richer contexts: time, activity, weather, and the ensemble effect of the recommendations list.

4. ACKNOWLEDGMENTS

The authors would like to thank Francesco Russo and Raffaele Teraoni Prioletti of Rai Research Centre for the discussions and support during the system implementation.

5. REFERENCES

[1] Radio rai. http://www.radio.rai.it. Accessed: 2016-11-15.

[2] G. Adomavicius and A. Tuzhilin. Context-aware recommender systems. In *Recommender systems handbook*, pages 191–226. Springer, 2015.

[3] J. I. Árnason, J. Jepsen, A. Koudal, M. R. Schmidt, and S. Serafin. Volvo intelligent news: A context aware multi modal proactive recommender system for in-vehicle use. *Pervasive and Mobile Computing*, 14:95 – 111, 2014. Special Issue on The Social Car: Socially-inspired Mechanisms for Future Mobility Services.

[4] M. Braunhofer, M. Kaminskas, and F. Ricci. Location-aware music recommendation. *International Journal of Multimedia Information Retrieval*, 2(1):31–44, 2013.

[5] M. Braunhofer, F. Ricci, B. Lamche, and W. Wörndl. A context-aware model for proactive recommender systems in the tourism domain. In *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services Adjunct*, MobileHCI '15, pages 1070–1075, New York, NY, USA, 2015. ACM.

[6] P. Casagranda, A. Erk, S. O'Halpin, D. Born, and W. Huijten. A framework for a context-based hybrid content radio. In *International Broadcasting Convention (IBC)*, 2015.

[7] P. Casagranda, M. L. Sapino, and K. S. Candan. Audio assisted group detection using smartphones. In *Multimedia & Expo Workshops (ICMEW), 2015 IEEE International Conference on*, pages 1–6. IEEE, 2015.

[8] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231, 1996.

[9] ETSI. Ts 103 270, RadioDNS hybrid radio; hybrid lookup for radio services, January 2015. http://www.etsi.org/deliver/etsi_ts/103200_103299/103270/01.01.01_60/ts_103270v010101p.pdf.

[10] S. Metta, P. Casagranda, A. Messina, M. Montagnuolo, and F. Russo. Leveraging MPEG-21 user description for interoperable recommender systems. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1072–1074. ACM, April 2016.

[11] M. Sarwat, J. J. Levandoski, A. Eldawy, and M. F. Mokbel. LARS*: An efficient and scalable location-aware recommender system. *IEEE Transactions on Knowledge and Data Engineering*, 26(6):1384–1399, 2014.

[12] D. Villeneuve. Lockito. https://play.google.com/store/apps/details?id=fr.dvilleneuve.lockito. Accessed: 2016-11-15.

[13] W. Woerndl, J. Huebner, R. Bader, and D. Gallego-Vico. A model for proactivity in mobile, context-aware recommender systems. In *Proceedings of the fifth ACM conference on Recommender systems*, pages 273–276. ACM, 2011.

[14] H. Yin, Y. Sun, B. Cui, Z. Hu, and L. Chen. LCARS: a location-content-aware recommender system. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 221–229. ACM, 2013.

In Search for Relevant, Diverse and Crowd-screen Points of Interests

Xiaoyu Ge
University of Pittsburgh
xig34@cs.pitt.edu

Samanvoy Reddy Panati
University of Pittsburgh
srp71@pitt.edu

Konstantinos Pelechrinis
University of Pittsburgh
kpele@pitt.edu

Panos K. Chrysanthis
University of Pittsburgh
panos@cs.pitt.edu

Mohamed A. Sharaf
University of Queensland
m.sharaf@uq.edu.au

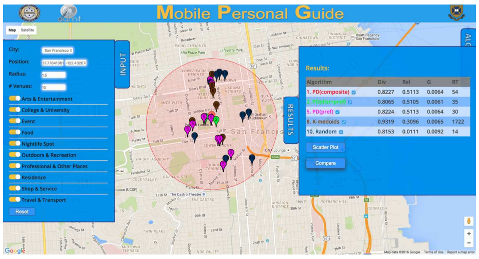
ABSTRACT

In this demo we present a prototype of an experimental platform for evaluating item recommendation algorithms. The application domain for our system is that of digital city guides. Our prototype implementation allows the user to explore different algorithms and compare their output. Among the algorithms implemented is MPG, which aims at providing a diverse set of recommendations better aligned with user preferences. MPG takes into consideration the user preferences (e.g., reach willing to cover, types of venues interested in exploring etc.), the popularity of the establishments as well as their distance from the current location of the user by combining them into a single composite score. We provide a web interface, which outputs on a map the recommended locations along with metadata (e.g., type and name of location, relevance and diversity scores, etc.). It also illustrates the potential of the Preferential Diversity approach on which MPG is based.

1. INTRODUCTION

The task of item recommendations is central to many applications in a variety of domains. At the core of these recommendation engines is a ranking of the items based on some quality features. The drawback of such an approach is that it does not allow for a **diverse** set of recommendations; similar items – with respect to some latent features – will tend to have similar rankings and hence, the top items will be similar to each other with high probability. Here diversity refers to latent attributes of the recommended items that cannot necessarily be captured by the single rating that the item has. This lack of diversity can further impact the *effective* choice set of the user, given that many of the recommended items will offer similar experiences.

In this work we develop a prototype system that serves as an experimental platform for exploring various approaches for the item recommendation problem. Our system is focused on the problem of recommending a set of venues to a user based on her current location and preferences. The system supports a number of different approaches for solving this recommendation problem, including our own algorithm, namely, Mobile Personal Guide (MPG), based on Preferential Diversity (*PrefDiv*) [2]. The platform developed (Fig. 1) allows us to compare the output of different recommendation engines, both visually (i.e., by presenting the recommended venues on a map) as well as based on traditional evaluation metrics (i.e., through a summary dashboard).



City: San Francisco
Position: 37.77641361 -122.432670
Radius: 1.5
Venues: 10

Results:

Algorithm	Div	Rel	G	RT
1. PD(composite) ☑	0.8227	0.5113	0.0064	54
2. PD(dist+pref) ☑	0.8065	0.5105	0.0061	35
5. PD(pref) ☑	0.8224	0.5113	0.0064	30
8. K-medoids ☑	0.9319	0.3096	0.0065	1722
10. Random ☑	0.8153	0.0111	0.0092	14

Figure 1: Our interface allows for experimenting with and comparing different recommendation algorithms.

Our current implementation includes the well known algorithms DisC Diversity [4], K-Medoid and a PageRank-based recommendation engine, as well as *PrefDiv*'s variations used in the MPG system [3]. While we have implemented the same diversity scheme for all the algorithms, our implementation is flexible and allows for different diversity and indexing schemes. Our prototype system is built using Java and the Google Maps API.

2. BACKGROUND

In this section, we introduce central concepts for the system.

Relevance: We represent the degree or score of relevance of an item o to a user u by the *Preference Intensity Value* (I_u^o).

Definition 1. *A Preference Intensity Value (I) is a decimal value used to express a negative preference* $[-1, 0)$*, a positive preference* $(0, 1]$*, or equality/indifference using* 0.

Diversity: We capture the diversity of a set of items S by computing the dissimilarity, measured through a semantic distance measure, of the pairs of items in S.

Definition 2. *Let S be the set of items. Two objects o_i and $o_j \in S$ are dissimilar to each other $dsm_\varrho(o_i, o_j)$, if $dt(o_i, o_j) > \varrho$ for some distance function dt and a real number ϱ, where ϱ is a distance parameter, which we call radius.*

Venue Flow Network: In our algorithms we will examine the integration of a flow network $\mathcal{G}_f$ between venues in a city as captured through the aggregate mobility of city-dwellers.

10.5441/002/edbt.2017.74

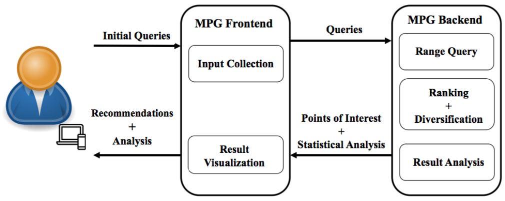


Figure 2: The system flow diagram of MPG.

DEFINITION 3. *The venue flow network $\mathcal{G}_f = (\mathcal{V}, \mathcal{E})$, is a directed network where a node $v_i \in \mathcal{V}$ represents a venue and there is a directed edge $e_{ij} \in \mathcal{E}$ from node v_i to node v_j, iff v_j has been visited immediately after v_i.*

$\mathcal{G}_f$ captures the aggregate mobility of dwellers and their transition flows across venues in the city. We integrate the PageRank $\boldsymbol{\pi}$ of $\mathcal{G}_f$ in the definition of a popularity-based intensity value for venue v.

3. SYSTEM DESIGN

Our experimental system consists of two modules (Fig.2); a back-end server (Sec. 3.1) and front-end interface (Sec. 3.2) that communicate through JSON. The back-end includes the implementation of the core recommender engines. The front-end interface (Fig.1) includes controls that allow the users to provide input parameters and obtain the queried recommendations.

3.1 Back-end Server

The problem at the epicenter of our experimental platform is formally defined as follows, where a point represents a venue (POI):

PROBLEM 1. *Given a set of geographical points $V = \{v_1, .., v_l\}$, a popularity index ξ_{v_i} for location v_i, a query point q, a reach r, and a profile set that encodes user preferences $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$, identify a set $V^* \subseteq V$ ($|V^*| = k$) with maximized diversity $\Delta(\mathcal{S})$, while a set of constraints $h(V^*, \mathcal{P}, q, r, \boldsymbol{\xi})$ is satisfied.*

The back-end currently implements and supports comparison among the following algorithms: DisC Diversity [4], K-Medoid, a PageRank-based recommendation engine, and *PrefDiv*'s variations proposed in the MPG system [3]. The *PrefDiv*'s variations differ in the way they compute the venues' intensity values used to rank the venues.

Range Queries: One of the main operation in the algorithms implemented in back-end server is to generate a nearest neighbor set. While several indexing schemes can be used, we utilize the *M-tree* spatial index structure [1] that has been used in DisC Diversity implementation [4]. M-tree is a balanced tree index that is designed to handle multi-dimensional dynamic data in general metric spaces, and it uses the triangle inequality for efficient range queries.

Ranking: MPG takes into consideration the user's preferences as captured through a hierarchical profile $\mathcal{P}$. The first level of $\mathcal{P}$ captures the preferences of the user expressed in terms of their (normalized) propensity to types of venues. The second layer of the user profiles further provides the propensity for specific establishments for the different types of venues. Fig. 3 presents a sample profile for a user. In our prototype implementations, the propensity values will be directly inputted by the user. However, in a real-world implementation these preferences can be inferred from historic data of visitations from the users (e.g., from the user's checkins on Foursquare).

MPG further defines a set of intensity values of the items, i.e., venues, to be recommended, based on the different *objectives* associated with Problem 1. For example, by considering the distance between the current location q of the user and venue v can also be used to obtain an intensity value for v. In particular with d_q^v being the normalized distance between q and v's location the distance-based intensity value can be defined as:

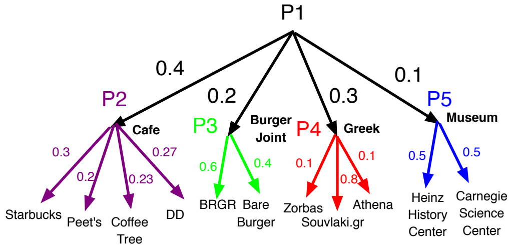


Figure 3: The first level of a user's profile corresponds to the coarse-grain preference profile (P_1), while each one of the subtrees stemming from P_1 corresponds to the preferences within each category (e.g., preference P_2 corresponds to the "Cafe" venue type).

$$I_d^v = 1 - \frac{d_q^v}{r} \tag{1}$$

In similar ways we can define a popularity-based intensity value I_p^v by considering the number of visitations to venue v. We can also incorporate additional popularity information by considering the Page Rank score π_v of venue v in the venue flow network. We also define a preference-based intensity value I_u^v. In our experimental system, we have implemented the computation of these intensity values as well as combinations of them, thereby providing a platform to compare between the different options (Table 1).

Diversification: The (dis)similarity between two venues is measured using two similarity distances: a syntactic distance based on the category structure of venues in Foursquare and a semantic distance based on the venue name.

Category Tree: The *category tree* is built to capture the category structure of venues in Foursquare. Each internal node in the category tree represents a type of venue, where each internal node represents the subcategory of the parent node with each leaf node representing the actual venue. There are in total 10 categories at the top-level of this hierarchy. Each internal node in a category tree contains the following attributes: ID of the category it represents, name of the category, a pointer to the parent node and a list of pointers to each of its children nodes. Since the degree of a node in the category tree is not bound, all the children node pointers are stored as hash tables, with the venue ID as the key and the pointer as the value.

The category tree can then be used to calculate the similarity distance between two venues v_i and v_j as follows:

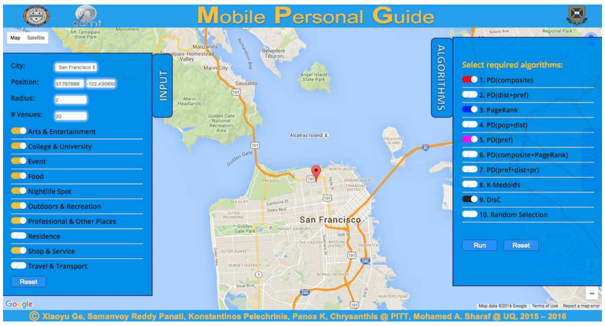
$$Similarity(v_i, v_j) = 1 - \frac{Ancestors\_Path}{Longest\_Path} \tag{2}$$

where *Ancestors_Path* is the number of common ancestors between the venues v_i and v_j and *Longest_Path* is the number of nodes on the longest path to the root from either v_i and v_j.

Word2Vec: Although the category tree is able to measure the similarity between two venues, this measurement is not very accurate as it cannot distinguish the difference between two venues that are under the same subcategory. In order to overcome this limitation, MPG utilizes the Word2Vec framework [5], an advanced NLP technique. Its word vector representation captures many linguistic regularities, and its computing model is based on the Neural Net Language model and more specifically the *Continuous Bag-of-Words* model (CBOW), which predicts the current word based on the sourcing words to generate all word vectors. The difference between two words under Word2Vec are calculated through the cosine similarity

Table 1: MODEL ABBREVIATION

Models	Description
PD(pref)	Uses preference-based intensity value as the relevance score for PrefDiv.
PD(pop+dist)	Uses popularity and distance from the user current location as the relevance score for PrefDiv.
PD(pref+dist+pr)	Uses preference-based intensity value, distance and PageRank as the relevance score for PrefDiv.
PD(dist+pref)	Uses preference-based intensity value and distance as the relevance score for PrefDiv.
PD(composite+PageRank)	Uses composite intensity value and PageRank as the relevance score for PrefDiv.
PD(composite)	Uses composite intensity value as the relevance score for PrefDiv.
PageRank	Only uses the result of PageRank as the final ranking without using PrefDiv.
DisC	Uses diversification method DisC [4] to generate recommendations, no PrefDiv involved.
K-medoids	Generate recommendations based on K-medoids clustering.
Random Selection	Uniformly select k items from all venues that with in the given radius from the query location.


Mobile Personal Guide
Map
Satellite
INPUT
ALGORITHMS
Reset
Run
Reset
San Francisco
Google
Map data ©2016 Google Terms of Use Report a map error
© Xiaoyu Ge, Samanvoy Reddy Panati, Konstantinos Pelechrinis, Panos K. Chrysanthis @ PITT, Mohamed A. Sharaf @ UQ, 2015 – 2016

City: San Francisco
Position: 37.797666 -122.430658
Radius: 2
Venues: 20

● Arts & Entertainment
● College & University
● Event
● Food
● Nightlife Spot
● Outdoors & Recreation
● Professional & Other Places
○ Residence
● Shop & Service
○ Travel & Transport

Select required algorithms:
● 1. PD(composite)
○ 2. PD(dist+pref)
● 3. PageRank
○ 4. PD(pop+dist)
● 5. PD(pref)
○ 6. PD(composite+PageRank)
○ 7. PD(pref+dist+pr)
○ 8. K-Medoids
● 9. DisC
○ 10. Random Selection

Figure 4: The user specifies the type of venues she wants to visit, i.e., the query input, and the algorithms to be used.

of two-word vectors.

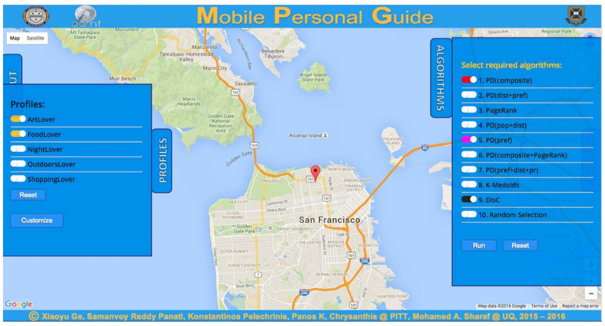
The current word vectors we adopted support phrases that consist of up to two words. For venue names that have more than two words or are not contained in the word vectors, we split the phrases into single words and then obtain word vectors for each individual word in the phrases. The final vector of a phrase is obtained through the average of all vectors for each word in this phrase. Since the accuracy of Word2Vec is strongly dependent on the quality of the word vectors, a large real-world corpus is needed in order to obtain high quality word vectors. The best suitable word vectors we obtained were generated from the entire English Wikipedia that consists of 55 GB of plain text. The resulting word vectors contain over 4 million entries. In order to effectively query the word vectors, MPG stores all the word vectors in memory as a hash map.

Our back-end server combines all of the above components and delivers relevant yet diverse recommendations to the user through the front-end described in the following section.

3.2 Front-end Interface

The front-end is implemented using the Google Maps API for visualizing the results on a map. It currently supports the cities of New York and San Francisco. The recommended points of interests (POIs) are numbered and colored to match the number and the color of the algorithm making the recommendation.

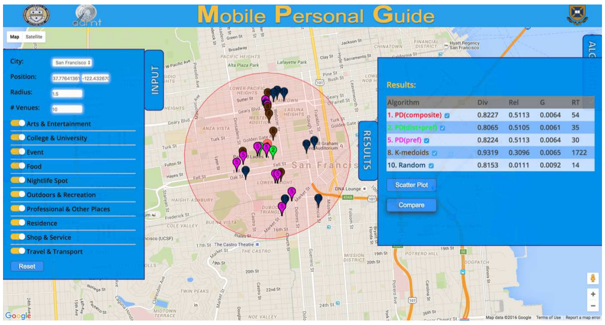
The interface consists of four different panels, namely, “Input”, “Algorithms”, “Profile” and “Results” (see Figs.4-6). The four first panels provide the options of selecting or setting the input parameters, the user profile, the recommendation algorithm(s), respectively. The last one shows the performance characteristics of the selected algorithms in tabular form as well as in a scatterplot. The listed characteristics in terms of quality are the relevance score of the selected venues, their diversity and the radius of gyration for the recommended set. We also report the run time taken for each algorithm as an indicator of interactivity.


Mobile Personal Guide
Map
Satellite
UT
PROFILES
ALGORITHMS
Reset
Customize
Run
Reset
San Francisco
Google
© Xiaoyu Ge, Samanvoy Reddy Panati, Konstantinos Pelechrinis, Panos K. Chrysanthis @ PITT, Mohamed A. Sharaf @ UQ, 2015 – 2016

Profiles:
● ArtLover
● FoodLover
○ NightLover
○ OutdoorsLover
○ ShoppingLover

Select required algorithms:
● 1. PD(composite)
○ 2. PD(dist+pref)
○ 3. PageRank
○ 4. PD(pop+dist)
● 5. PD(pref)
○ 6. PD(composite+PageRank)
○ 7. PD(pref+dist+pr)
○ 8. K-Medoids
● 9. DisC
○ 10. Random Selection

Figure 5: The user has the ability to choose one of the pre-loaded profiles for the type of venues she is looking for.



City: San Francisco
Position: 37.77641361 -122.432670
Radius: 1.5
Venues: 10

● Arts & Entertainment
● College & University
● Event
● Food
● Nightlife Spot
● Outdoors & Recreation
● Professional & Other Places
● Residence
● Shop & Service
● Travel & Transport

Results:

Algorithm	Div	Rel	G	RT
1. PD(composite) ☑	0.8227	0.5113	0.0064	54
2. PD(dist+pref) ☑	0.8065	0.5105	0.0061	35
5. PD(pref) ☑	0.8224	0.5113	0.0064	30
8. K-medoids ☑	0.9319	0.3096	0.0065	1722
10. Random ☑	0.8153	0.0111	0.0092	14

Figure 6: Results are displayed on a map, while also providing a numerical comparison of the chosen algorithms as well.

4. DEMONSTRATION PLAN

During the demonstration, we will run the front-end interface of the system on one or more laptops and the backend would be hosted on a remote sever. The participants will be have the opportunity to interact in different modes, that of an application end-user and of an experimental researcher.

Application end-user: In this scenario, attendees will experience the effectiveness of our system through the view of an ordinary user. Specifically, they are able to provide the initial location (i.e., coordinates) for their POIs recommendation query (Fig.7), while and they can use the “Input” panel to provide additional information for the query, i.e., radius willing to cover, types of venues interested in ex-

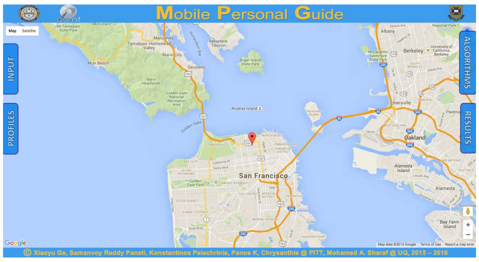
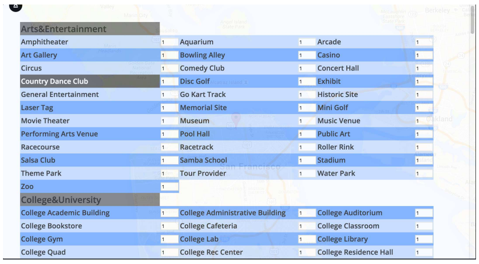

Mobile Personal Guide
INPUT
PROFILES
ALGORITHMS
RESULTS
San Francisco
Oakland
Berkeley
Alameda
Google
© Xiaoyu Ge, Samanvoy Reddy Panati, Konstantinos Pelechrinis, Panos K, Chrysanthis @ PITT, Mohamed A. Sharaf @ UQ, 2015 – 2016

Figure 7: The user can mark her location of interest.



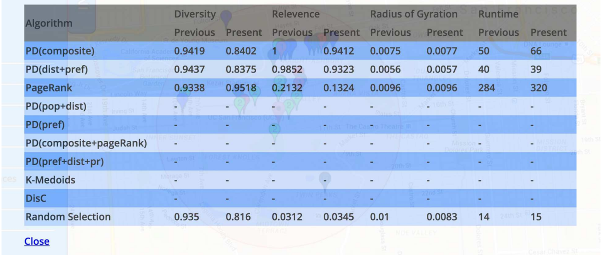
Arts&Entertainment					
Amphitheater	1	Aquarium	1	Arcade	1
Art Gallery	1	Bowling Alley	1	Casino	1
Circus	1	Comedy Club	1	Concert Hall	1
Country Dance Club	1	Disc Golf	1	Exhibit	1
General Entertainment	1	Go Kart Track	1	Historic Site	1
Laser Tag	1	Memorial Site	1	Mini Golf	1
Movie Theater	1	Museum	1	Music Venue	1
Performing Arts Venue	1	Pool Hall	1	Public Art	1
Racecourse	1	Racetrack	1	Roller Rink	1
Salsa Club	1	Samba School	1	Stadium	1
Theme Park	1	Tour Provider	1	Water Park	1
Zoo	1				
College&University					
College Academic Building	1	College Administrative Building	1	College Auditorium	1
College Bookstore	1	College Cafeteria	1	College Classroom	1
College Gym	1	College Lab	1	College Library	1
College Quad	1	College Rec Center	1	College Residence Hall	1

Figure 8: The user can tweak the pre-loaded profiles by altering the preference values at the various venue types.

ploring etc. (Fig.4). Then, attendees can use the "Profile" panel to select a preference profile out of a set of predefined ones such as ArtLover, FoodLover and OutdoorsLover (Fig.5). Finally, attendees can choose one or more algorithm among the different ones currently implemented (see Table 1) through the "Algorithms" menu (Fig.4). Once the algorithms for the experiment are chosen, they can submit the POIs recommendation query for execution via the "Algorithms" panel. The POIs returned will be visualized on a map and color-coded based on the algorithm used for making the recommendation (Fig.6).

Experimental researcher: In this scenario we will demonstrate the platform's ability to be used for exploration and comparison of the trade-offs between different parameter configurations and recommendation algorithms. This would enable the "expert" users (i.e., researchers) to explore the characteristic of different algorithms and parameters. Specifically, researchers can customize a selected preference profile by adjusting the values on the corresponding category sub-tree ("Customize" pop-up, Fig.8). Furthermore, researchers are provided with knobs for tuning the parameters that are used in calculating the composite intensity value $I^v_{p,d,u}$, which is obtained by combining the popularity intensity value I^v_p, the distance intensity value $I^v_{d,q}$ and the preference-based intensity value I^v_u.

Researchers can also choose to run multiple algorithms simultaneously, and take the advantage of a dashboard where results will be presented with respect to the relevance score of the selected venues, their diversity, the radius of gyration for the recommended set, as well as the run time taken for the algorithm ("Results" panel, Fig.6). The researcher also has the option to visualize the results on a scatterplot (Fig.10) that makes it straightforward to compare the various schemes. The "Results" panel also has an option called "compare," which enables the user to compare the results of the present query with the previous one (Fig.9). The researcher can further explore and compare other algorithms by choosing them from the "Results" panel. This will simply overlay the new results over the existing ones.



Algorithm	Diversity		Relevence		Radius of Gyration		Runtime	
	Previous	Present	Previous	Present	Previous	Present	Previous	Present
PD(composite)	0.9419	0.8402	1	0.9412	0.0075	0.0077	50	66
PD(dist+pref)	0.9437	0.8375	0.9852	0.9323	0.0056	0.0057	40	39
PageRank	0.9338	0.9518	0.2132	0.1324	0.0096	0.0096	284	320
PD(pop+dist)	-	-	-	-	-	-	-	-
PD(pref)	-	-	-	-	-	-	-	-
PD(composite+pageRank)	-	-	-	-	-	-	-	-
PD(pref+dist+pr)	-	-	-	-	-	-	-	-
K-Medoids	-	-	-	-	-	-	-	-
DisC	-	-	-	-	-	-	-	-
Random Selection	0.935	0.816	0.0312	0.0345	0.01	0.0083	14	15

Close

Figure 9: The user can navigate and compare the current recommendations with those of the previous setting.

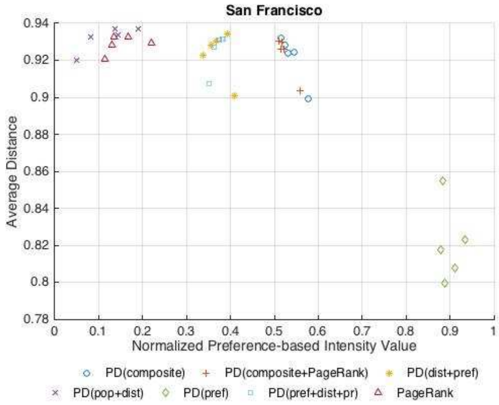


Figure 10: An option for visualizing the performance evaluation of the chosen algorithms is also available to the user.

5. CONCLUSIONS

In this paper, we presented a prototype platform that allows users to experiment with different recommendation schemes that aim at providing a diverse, yet relevant to the user preferences, set of objects. Our prototype allows the implementation and experimentation with new recommender algorithms (e.g., ranking and similarity schemes) as well as different implementations of the various backend units (e.g., indexing).

As an experimental platform, we have utilized a static dataset collected from Foursquare's API. However, in a real-world application, using static datasets will certainly affect the quality of recommendations since it will be based on possibly stale information. As a real-world application the system must pull the data needed for providing recommendations either periodically (e.g., once a day) or in real-time, e.g., in order to utilize how many people are checked-in at a given time at a venue.

6. REFERENCES

[1] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric spaces. In *VLDB*, 1997.

[2] X. Ge, P. K. Chrysanthis, and A. Labrinidis. Preferential Diversity. In *ExploreDB*, 2015.

[3] X. Ge, P. K. Chrysanthis, and K. Pelechrinis. MPG: Not so Random Exploration of a City. In *IEEE MDM*, 2016.

[4] E. P. Marina Drosou. Multiple Radii DiSC Diversity: Result Diversification based on Dissimilarity and Coverage. *ACM TODS*, 40(1), 2015.

[5] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Distributed Representations of Words and Phrases and their Compositionality. In *NIPS*, 2013.

SIMDMS: Data Management and Analysis to Support Decision Making through Large Simulation Ensembles [*]

Silvestro Poccia Maria Luisa Sapino
University of Torino, Italy
{poccia,mlsapino}@di.unito.it

Sicong Liu Xilun Chen Yash Garg Shengyu Huang
Jung Hyun Kim Xinsheng Li Parth Nagarkar K. Selçuk Candan
Arizona State University, USA
{ s.liu,xilun.chen,ygarg,shuang54,jkim294,lxinshen,pnagarka,candan}@asu.edu

ABSTRACT

Data- and model-driven computer simulations are increasingly critical in many application domains. These simulations may track 100s or 1000s of inter-dependent parameters, spanning multiple layers and spatial-temporal frames, affected by complex dynamic processes operating at different resolutions. Because of the size and complexity of the data and the varying spatial and temporal scales at which the key processes operate, experts often lack the means to analyze results of large simulation ensembles, understand relevant processes, and assess the robustness of conclusions driven from the resulting simulations. Moreover, data and models dynamically evolve over time requiring continuous adaptation of simulation ensembles. The `simDMS` platform aims to address the key challenges underlying the creation and use of large simulation ensembles and enables (a) execution, storage, and indexing of large ensemble simulation data sets and the corresponding models; and (b) search, analysis, and exploration of ensemble simulation data sets to enable ensemble-based decision support.

Keywords

Simulation ensembles, multivariate time series

1. INTRODUCTION

Data- and model-driven computer simulations are increasingly critical in many application domains.

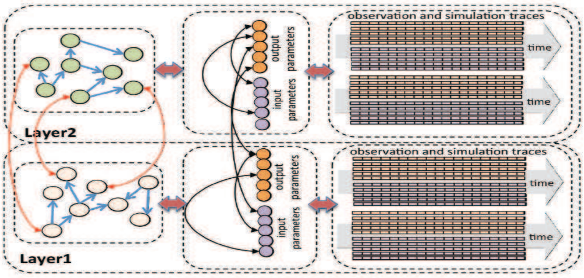


Figure 1: Simulation ensembles are (a) multivariate, (b) multi-modal (temporal, spatial, hierarchical, graphical), (c) multi-layer, (d) multi-resolution, and (e) inter-dependent (i.e., observations of interest depend on and impact each other)

Epidemic Simulation Ensembles: For example, for predicting geo-temporal evolution of epidemics and assessing the impact of interventions, experts often rely on epidemic spread simulation software such as (e.g., GLEaM [2] and STEM [3]). The GLEaM simulation engine, for example, consists of three layers: (a) a population layer, (b) a mobility layer which includes both long-range air travel and short-range commuting patterns between adjacent subpopulations, and (c) an epidemic layer which allows the user to specify parameters (such as reproductive number and seasonality) for the infectious disease, initial outbreak conditions (e.g. seeding of the epidemic and the immunity profile of the subpopulation), and intervention measures.

Building Energy Simulation Ensembles: Similarly, effective building energy management, leading to more sustainable building systems and architectural designs with monitoring, prioritization, and adaptation of building components and subsystems, requires large data-driven simulations involving (a) location and climate information for the city in which the building is located, (b) building construction information, such as building geometry and surface constructions (including exterior walls, interior walls, partitions, floors, ceilings, roofs, windows and doors), (c) building use information, including the lighting and other equipment (e.g. electric, gas, etc.) and the number of people in each area of the building, (d) building thermostatic control information, including the temperature control strategy for each area, (e) heating, ventilation, and air conditioning (HVAC) operation

[*] Research is supported by NSF#1318788 "Data Management for Real-Time Data Driven Epidemic Spread Simulations", NSF#1339835 "E-SDMS: Energy Simulation Data Management System Software", NSF#1610282 "DataStorm: A Data Enabled System for End-to-End Disaster Planning and Response", NSF#1633381 "BIGDATA: Discovering Context-Sensitive Impact in Complex Systems", and "FourCmodeling": EU-H2020 Marie Sklodowska-Curie grant agreement No 690817.

Series ISSN: 2367-2005 10.5441/002/edbt.2017.75

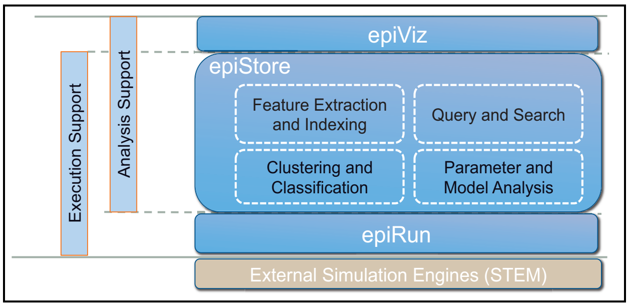

epiViz
epiStore
Feature Extraction and Indexing
Query and Search
Clustering and Classification
Parameter and Model Analysis
epiRun
External Simulation Engines (STEM)
Execution Support
Analysis Support

Figure 2: simDMS system overview (instantiated with epidemic simulation ensembles)

and scheduling information, and (f) central plant information for specification and scheduling of boilers, chillers, and other equipment. EnergyPlus software, for example, relies on the description of the building's physical make-up and associated mechanical and other systems and includes time-step based simulation for many energy-related building parameters [1].

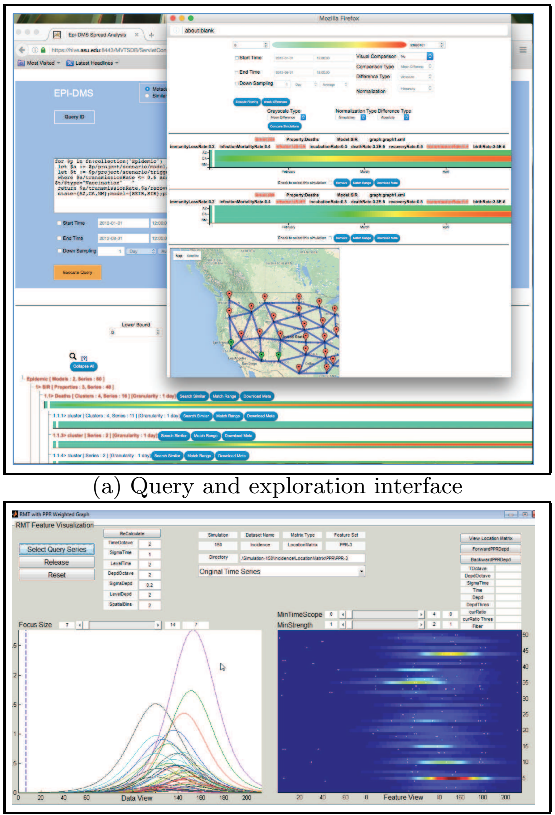
1.1 Challenge: Ensemble based Decisions

While, in most cases, very powerful simulation software exist, using these simulation software for decision making faces several significant challenges: (a) *Creating correct simulation models is a costly operation*, and it is often the case that the designed simulation models are incomplete or imprecise. (b) Also, *the execution of a simulation can be very costly*, given the fact that complex, inter-dependent parameters affected by complex dynamic processes at varying spatial and temporal scales have to be taken into account. (c) *A third major source of cost is the simulation ensemble analysis*: because of the size and complexity of the data and the varying spatial and temporal scales at which the key processes operate, experts often lack the means of analyzing results of large simulation ensembles, understanding relevant processes, and assessing the robustness of conclusions driven from the resulting simulations.

As visualized in Figure 1, the key characteristics of the simulation data sets include the following: (a) multi-variate, (b) multi-modal (temporal, spatial, hierarchical, graphical), (c) multi-layer, (d) multi-resolution, and (e) inter-dependent (i.e., observations of interest depend on and impact each other). In particular, simulations may track 100s or 1000s of inter-dependent parameters, spanning multiple layers and spatial-temporal frames, affected by complex dynamic processes operating at different resolutions. Moreover, generating an appropriate *ensemble* of stochastic realizations may require multiple simulations, each with different parameter settings corresponding to slightly different, but plausible, scenarios. As a consequence, running simulations and interpreting simulation results (along with the real-world observations) to generate timely actionable results are difficult.

We argue that these challenges can be significantly alleviated using a data-driven approach that addresses the following fundamental questions:

- Given a large parameter space and fixed budget of simulations, can we decide which simulations to execute in the ensemble? Can we revise the ensemble as we receive a stream of real world observations?
- Can we compare a large number of simulation ensembles and observations (under different parameter settings) to identify their similarities and differences? Can we analyze one or more simulation ensembles to discover patterns and relationships between input parameters, key events/interventions, and simulation outcomes? Can we discover key events and summarize a large simulation ensemble to highlight these events? Can we classify these key events?
- Can we search and explore simulation ensembles based on the underlying key events or the overall simulation similarities? Can we keep track of the most relevant and most outlier simulations in an ensemble as we receive a stream of real world observations?



(a) Query and exploration interface

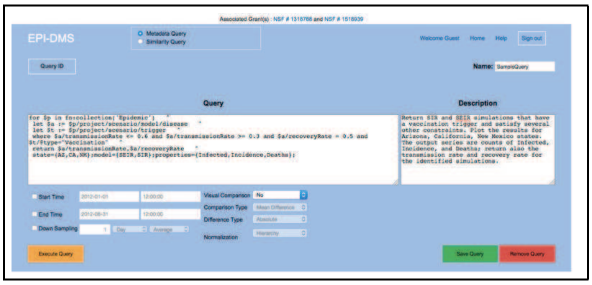
(b) Simulation visualization interface

Figure 3: simVIZ simulation query, visualization, and analysis interfaces (instantiated with epidemic simulation ensembles): visualizing an epidemic simulation as a multi-variate time series and the key robust multivariate (RMT) events [9] identified on a given simulation

1.2 simDMS Overview

The simDMS system (Figure 2) and its visualization engine simVIZ (Figure 3) aim at assisting users to explore large simulation ensembles while limiting the impact of aforementioned challenges [4, 5, 6, 7, 8]. In particular, simDMS supports

- *analysis and indexing of simulation data sets,* including extraction of salient multi-variate temporal features from inter-dependent parameters (spanning multiple layers and spatial-temporal frames, driven by complex dynamic processes operating at different resolutions) and indexing of these features for efficient and accurate search and alignment;



Name: SampleQuery

```
for $p in fn:collection('Epidemic')   ^
 let $a := $p/project/scenario/model/disease   ^
 let $t := $p/project/scenario/trigger   ^
 where $a/transmissionRate <= 0.6 and $a/transmissionRate >= 0.3 and $a/recoveryRate = 0.5 and
$t/@type="Vaccination"   ^
 return $a/transmissionRate,$a/recoveryRate   ^
 state={AZ,CA,NM};model={SEIR,SIR};properties={Infected,Incidence,Deaths};
```

Return SIR and SEIR simulations that have a vaccination trigger and satisfy several other constraints. Plot the results for Arizona, California, New Mexico states. The output series are counts of Infected, Incidence, and Deaths; return also the transmission rate and recovery rate for the identified simulations.

☐ Start Time 2012-01-01 12:00:00
☐ End Time 2012-08-31 12:00:00
☐ Down Sampling 1 Day Average
Visual Comparison: No
Comparison Type: Mean Difference
Difference Type: Absolute
Normalization: Hierarchy

```
FOR $p in fn:collection('EpidemicSimulationEnsemble') ^
LET $diseaseModel := $p/project/scenario/model/disease     ^
LET $triggerModel := $p/project/scenario/trigger      ^
LET $epidemicScenario := $p/project/scenario ^
WHERE
 $diseaseModel/transmissionRate <= 0.6 and
 $diseaseModel/transmissionRate >= 0.3 and
 $diseaseModel/recoveryRate = 0.5 and
 $triggerModel/@type="Vaccination" and
 ($epidemicScenario/infector/@targetISOKey="US-CA"  or
  $epidemicScenario/infector/@targetISOKey="US-NY" ) and
 ($epidemicScenario/graph = "mobility_graph_7.xml"  or
  $epidemicScenario/graph = "mobility_graph_8.xml")  ^
RETURN
 $diseaseModel/transmissionRate,
 $diseaseModel/recoveryRate,
 $epidemicScenario/graph   ^
 STATE={AZ,CA,NM};
 MODEL={SEIR,SIR};
 PROPERTIES={Infected,Incidence,Deaths};
 FROM ={01/01/2012 12:00:00}; TO={08/31/2012 12:00:00};
 BY={1-D}; FUNCTION ={avg};
```

Figure 4: A metadata query over an epidemic simulation ensemble

- *parameter and feature analysis,* including identification of unknown dependencies across the input parameters and output variables spanning the different layers of the observation and simulation data. These, and the processes they imply, can be used for understanding and refining the parameter dependencies and models.

Query and visualization interfaces for the epidemic (epiDMS) and building energy (eDMS) instantiations of the simDMS platform can be found at http://aria.asu.edu/epidms and http://aria.asu.edu/edms, respectively. You can watch a tutorial at https://youtu.be/9w-4nDhXv3k .

2. DEMONSTRATION SCENARIOS

We will demonstrate the system on (a) epidemic simulation data sets created using the Spatiotemporal Epidemiological Modeler (STEM) [3] and (b) building energy simulation data sets, created using the EnergyPlus building energy simulation program [1]. The simulations will be stored in simDMS and will be visually analyzed during the demonstration using simVIZ.

2.1 Simulation Ensemble Planning

A simulation ensemble (consisting of a set of simulation instances sampled from an input parameter space) can be seen as defining an outcome-surface for each of the output variables (such as the number of deaths that will result from an epidemic): each outcome-surface describes the probability distribution of the potential outcomes for the corresponding variable. These simulation ensembles, consisting of potentially tens of thousands of simulations, are expensive to obtain: therefore we need sampling strategies for the input parameter spaces that eliminate irrelevant scenarios in such a way that more accurate simulation results are obtained where they are more relevant. Moreover, these simulation ensembles need to be continuously revised and refined as the situation on the ground changes: (a) revisions involve incorporating real-world observations into existing simulations to alter their outcomes; (b) refinements involve identifying new simulations to run based on the changing situation on the ground. Therefore, we will demonstrate data-driven sampling strategies to decide (given a budget of simulations) which simulations to run and incremental non-uniform sample-based data construction techniques to revise outcome-surfaces. We will specifically highlight how to assign utility- and cost-functions for each potential sample (based on how well the observed data are fitting the previous simulations, how likely a new simulation at the given sample improves the accuracy of fit, and how costly the corresponding simulation would be) and use these functions to decide the optimal re-sampling strategy.

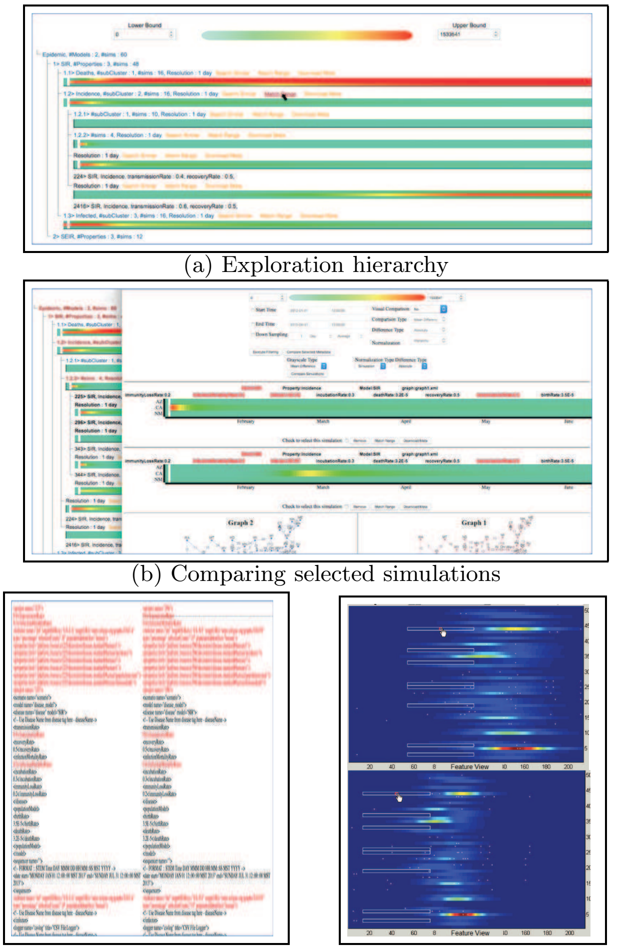
2.2 Scenario- and Similarity-based Querying

A basic function of the simDMS system is to retrieve simulations based on a user-specified scenario description. Figure 4 presents a sample query:

- The "FOR" statement allows the user to select the simulation dataset to query. In this example, the user focuses on the stored simulation set "EpidemicSimulationEnsemble".
- The "LET" statement allows the user to associate variables representing disease and intervention trigger models with epidemic scenarios.
- The "WHERE" clause allows the user to specify conditions on the simulation models to filter those simulations that are relevant for the current analysis. In this example, the user specifies that for the returned simulations, the transmission rate parameter should be between 0.3 and 0.6, the recovery rate parameter should be set to 0.5, and that a "vaccination" type trigger should be included in the simulation model. The user also specifies that epidemic should have started in California (CA) or New York (NY) and the "mobility_graph_7.xml" or "mobility_graph_8.xml" should have been used to generate the simulations.
- The "RETURN" clause lists the simulation parameters to be returned in the result. In this example, the user is interested in the transmission rate, recovery rate, the mobility graph for each returned simulation. In addition, the query asks the system to return the time series corresponding to the "infected", "incidence", and "deaths" simulation output parameters for Arizona (AZ), California (CA), and New Mexico (NM).
- The user further specifies that s/he is interested in only the first 8 months of the simulation.
- Finally, the user specifies that the system returns daily (1-D) averages of the simulation parameters for the specified duration.

Note that, in order to process this single query, simDMS combines data of different forms (structured, semi-structured, and temporal), stored in different back-end storage engines.

In addition to scenario-based filtering and search, the platform also enables searching and/or triggering based on particular temporal patterns on the ensembles. This feature



(a) Exploration hierarchy

(b) Comparing selected simulations

(c) Metadata comparison (d) Feature-based comparison

Figure 5: Sample interfaces for exploring ensembles

allows the expert to identify relevant subsets of stored simulations that match actual real-world observations or specific targets for intervention measures.

2.3 Analysis and Exploration of Ensembles

Once the query is executed and the relevant simulations are identified, the system then organizes the results into a navigable hierarchy, based on the temporal dynamics of the simulation results (Figure 5): Since simulation data sets can be viewed as multi-variate time series, simVIZ focuses on visual analysis (e.g. event detection, similarity and difference analysis) of single and multiple multi-variate simulation data sets. Scenarios that result in similar patterns are grouped under the same branch, while simulations that show major differences in disease development are placed under different branches of the navigation hierarchy. The user can then navigate this hierarchy using "drill-down" and "roll-up" operations and pick sets of simulations to study and compare the corresponding scenarios in further detail.

The interface presents both conventional series plots as well as heatmap visualizations, where each series is shown as a row of pixels. It is important to note that, while the temporal (i.e., horizontal) axis is ordered, the vertical axis corresponding to the different states is not ordered, in that two nearby states according to user mobility may not be neighboring rows on the interface due to the complexity of the mobility graph. The interface also highlights, on the heatmap, the major *robust multi-variate time series (RMT) features* (optimized for supporting alignments of multi-variate time series, leveraging known correlations and dependencies among the variates [9]) identified on the heatmap. An RMT feature is a part of the time series that is *different* in structure from its immediate context in time and/or variate relationships. A key property of these RMT features is that they are *robust* against noise and common transformations, such as temporal shifts or missing variates. This is illustrated in Figure 5(d), which shows two different epidemic simulations, with the same starting state, but different disease parameters and interventions. While the resulting disease evolutions are visibly different in shape, the same multi-variate feature (corresponding to the onset of the disease on the same nearby states) is identified on both simulations. This robustness property of RMT features enables various simVIZ functions, such as search, clustering, classification, and summarization of simulations and large simulation data sets [4, 5, 6, 7, 8].

3. CONCLUSIONS

The `simDMS` platform provides metadata and event-driven analysis and visualization of simulation ensembles to assist decision makers to query and explore ensemble simulations and decide which additional simulations to execute.

4. ACKNOWLEDGMENTS

We also thank Reece Bailey, Hans Behrens, Sarah Fallah-Adl, Ashish Gadarki, Anisha Gupta, Divyanshu Khare, Mao-Lin Li, Nancy Li, Samuel Morton, Shivam Sadakar, Xiaolan Wang, Fiona Zhang, Luke Zhang, and Jerry Zhu for their contributions to the `simDMS` platform.

5. REFERENCES

[1] EnergyPlus Energy Simulation Software. http://apps1.eere.energy.gov/buildings/energyplus/

[2] The Global Epidemic and Mobility Model, GLEAM. http://www.gleamviz.org

[3] STEM. The Spatiotemporal Epidemiological Modeler Project. Available at http://www.eclipse.org/stem.

[4] S. Huang, K.S. Candan, M.L.Sapino. BICP: Block-Incremental CP Decomposition with Update Sensitive Refinement. CIKM 2016: 1221-1230

[5] X. Li, S. Huang, K.S. Candan, M.L. Sapino. 2PCP: Two-phase CP Decomposition for Billion-Scale Dense Tensors. ICDE 2016: pp. 835-846.

[6] S. Liu, Y. Garg, K.S. Candan, M.L. Sapino, G. Chowell. NOTES2: Networks-Of-Traces for Epidemic Spread Simulations. AAAI Workshop on Computational Sustainability, 2015.

[7] S. Liu, S. Poccia, K.S. Candan, G. Chowell, and M.L. Sapino. epiDMS: Data Management and Analytics for Decision-Making From Epidemic Spread Simulation Ensembles. Journal of Infectious Diseases. 2016.

[8] P. Nagarkar, K.S. Candan, A. Bhat. Compressed Spatial Hierarchical Bitmap (cSHB) Indexes for Efficiently Processing Spatial Range Query Workloads. PVLDB 8(12): 1382-1393 (2015)

[9] X.Wang, K.S. Candan, and ML Sapino. Leveraging Metadata for Identifying Local, Robust Multi-Variate Temporal (RMT) Features. ICDE 2014, 388-399.

In-Place Appends for Real: DBMS Overwrites on Flash without Erase

Sergey Hardock
Databases and Distributed Systems Group
TU-Darmstadt, Germany
hardock@dvs.tu-darmstadt.de

Ilia Petrov
Data Management Lab
Reutlingen University, Germany
ilia.petrov@reutlingen-university.de

Robert Gottstein
Databases and Distributed Systems Group
TU-Darmstadt, Germany
gottstein@dvs.tu-darmstadt.de

Alejandro Buchmann
Databases and Distributed Systems Group
TU-Darmstadt, Germany
buchmann@dvs.tu-darmstadt.de

ABSTRACT

In the present paper we demonstrate a novel approach to handling small updates on Flash called In-Place Appends (IPA). It allows the DBMS to revisit the traditional write behavior on Flash. Instead of writing whole database pages upon an update in an out-of-place manner on Flash, we transform those small updates into update deltas and append them to a reserved area on the very same physical Flash page. In doing so we utilize the commonly ignored fact, that under certain conditions Flash memories can support in-place updates to Flash pages without a preceding erase operation.

The approach was implemented under Shore-MT and evaluated on real hardware. Under standard update-intensive workloads we observed 67% less page invalidations resulting in 80% lower garbage collection overhead, which yields a 45% increase in transactional throughput, while doubling Flash longevity at the same time. The IPA outperforms In-Page Logging (IPL) by more than 50%.

We showcase a Shore-MT based prototype of the above approach, operating on real Flash hardware – the OpenSSD Flash research platform. During the demonstration we allow the users to interact with the system and gain hands-on experience of its performance under different demonstration scenarios. These involve various workloads such as TPC-B, TPC-C or TATP.

1. INTRODUCTION

A well-known property of Flash memory is the erase-before-overwrite principle. In order to update the content of a certain Flash page, the corresponding Flash block must be erased first and all valid pages must be written back. Since this results in huge I/O latencies and rapid wear-out, all modern SSDs utilize some variant of an out-of-place update strategy. The updated Flash pages are always written to a new physical location, while the old pages are simply invalidated and the occupied space eventually gets reclaimed by the garbage collection (GC). Although, this allows postponing expensive erases and page migrations and executing them in the background, the *on-device write-amplification* produced by the GC is a major performance bottleneck of modern Flash SSDs [4]. Another source of the write-amplification in traditional DBMSs are the *write behavior* and *I/O granularity*. Regardless of the size of the updated information on a database page, the whole page is written out to stable storage (Figure 1). Our analysis of the standard OLTP benchmarks (TPC-B/-C and TATP), as well as social network workload based on LinkBench has shown that in more than 70% of evicted dirty 8KB-pages, less than 100 bytes of net data is modified. Thus, for 100 modified bytes in total the DBMS writes out the whole 8KB database pages. This results in the *DBMS write-amplification* (ratio of written and actually changed bytes) of about 80x. The file system underneath can further increase this value [9].

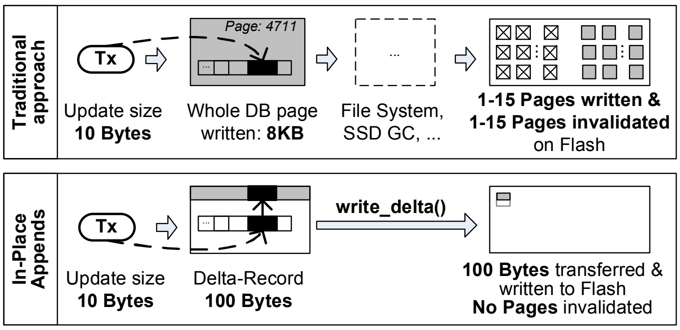


Figure 1: Write-amplification: traditional vs IPA.

To handle both kinds of write-amplification on Flash we proposed an approach called *In-Place Appends (IPA)* [5]. Its basic idea is to transform small in-place updates performed by DBMS transactions into delta-records upon page eviction. Furthermore, those delta-records are appended to a reserved area on the *very same physical Flash pages* along with the original content. In doing so we utilize the commonly ignored fact that under certain conditions physical Flash pages can be updated in-place without a

Series ISSN: 2367-2005 10.5441/002/edbt.2017.76

preceding erase operation. By relaxing this *erase-before-overwrite principle* we can significantly reduce the number of page invalidations and out-of-place updates. Furthermore we reduce the GC overhead (page migrations and erase operations) and achieve lower I/O latencies. Additionally, the *DBMS write-amplification* is reduced by a newly defined command *write_delta*, which allows the DBMS to write out only the delta-records instead of whole pages.

The IPA [5] was implemented in Shore-MT. Although the IPA is well applicable to traditional black-box SSD architectures, we have implemented it as an extension of open NoFTL architecture [6], due to the clear performance advantages of the latter. The NSM page layout was accordingly modified to "accommodate" the delta-record area, while buffer and storage management took the responsibility for creating and applying of delta-records for page reconstruction. The use of *NoFTL regions* [7] allows applying IPA selectively, only to certain database objects that are dominated by small-sized updates. The evaluation is performed on the OpenSSD Jasmine hardware: a research SSD platform with programmable controller and MLC Flash modules. Throughout the experiments under standard OLTP workloads (TPC-C, TPC-B and TATP) we observed up to 45% improvement of transactional throughput by performing up to 80% less page migrations and erase operations as compared to the traditional approach. Besides the clear performance advantages, the reduction of GC overhead results in doubling the longevity of Flash SSD.

In-Page Logging [8] is a well-known approach and the closest competitor of IPA. A major difference to IPA is the way the delta-records (or update logs in IPL) are persisted. IPL writes out the update logs either upon the page eviction or fullness of in-memory log buffer. The logs are written to the separate, reserved Flash pages on the same Flash block the original data is. Thus, to reconstruct the up-to-date version of the database page multiple Flash pages must be read (Flash page(s) with the original data and the one or more Flash pages with update logs). Under modern OLTP workloads with 70% to 90% reads, doubling the read load causes significant performance bottlenecks. In contrast, IPA does not produce any additional read overhead, since delta-records are co-located with the original content on the same Flash page. Furthermore, IPA performs 23% to 62% less writes and 29% to 74% less erases as compared to IPL on a range of OLTP workloads.[1]

2. REVISITING ERASE-BEFORE-OVERWRITE PRINCIPLE

The elementary unit of Flash memory is a single Flash cell - a floating gate (or a charge trap in 3D NAND). The cells of each Flash block are connected in the form of a lattice (see Figure 2), where rows are known as wordlines and columns as bitlines. Cells of each wordline build one (SLC) or several (MLC) physical Flash pages. This physical layout of NAND Flash is optimized for the fast access to the whole Flash pages, since writing and reading is done on per wordline basis.

It is worth, however, to look deeper into the write process on the Flash. To program a Flash page, at first, the corresponding wordline (e.g. WL30 on the Figure 2) is selected by applying a high voltage (e.g. 20V) to it. Then, depending on the value of each bit of data being programmed, the voltages on the corresponding bitlines are selected respectively. Thus, for instance, by applying the VCC voltage to a bitline, the corresponding cell on selected wordline is left unprogrammed, i.e. no charge is "inserted" into this cell, while by applying 0V voltage the corresponding cell will be programmed to a certain charge. Further, the programming of each particular cell is done in multiple steps. This technique is applied by all modern Flash SSDs and is known as Incremental Step Pulse Programming (ISPP) [3]. The charge of programmed cells is increased incrementally in small "portions", while after each programming iteration the cell is sensed (read) to check if the desired charge level is achieved. It is important to note, that to increase the charge of any individual cell no foregoing erase operation is required. Only if the charge level needs to be decreased - the whole corresponding Flash block must be erased (i.e. all cells are reseted). The probability that a random update on a Flash page results *only* in increase of the charge levels of corresponding cells[2] is negligibly small. *Therefore, in the common case the updates can not be performed in-place (erase-before-overwrite principle).*

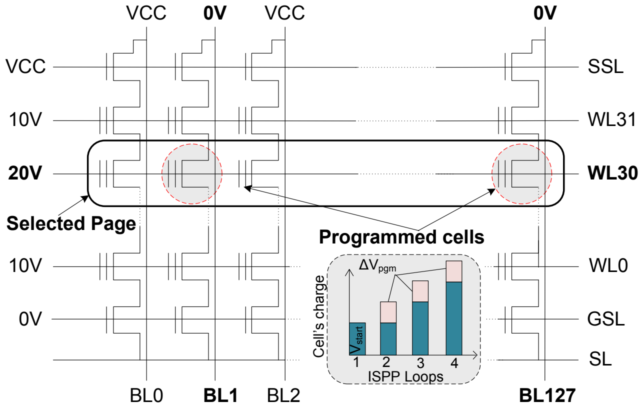


Figure 2: Organization of SLC NAND Flash memory and ISPP.

But what if an update on a Flash page is performed in the form of an append? Assume, for instance, the 8KB Flash page is programmed initially with only 6KB data. In this case, the cells that correspond to the remaining 2KB are left unprogrammed. Now, the original 6KB of data are augmented with the 2KB of new data. *This new version of the page (original data & append) can actually be written (programmed) in-place, i.e. by "overwriting" the append area of the original Flash page without foregoing erase operation.* This is possible because all newly programmed cells only increase their charge. The existing charge within the cells storing the original data is left unchanged during the overwrite.

3. BRIEF OVERVIEW OF IN-PLACE APPENDS

The main idea is to transform small-size updates on DB pages into delta-records upon page eviction from the buffer pool. The delta records are then appended to the reserved space on a page, so-called delta-record area, while the original content of the page is left unchanged. By doing so, the database page can be written to the *very same physical Flash address* without page invalidation or foregoing erase operation. The major "points of attack" by the implementation of the approach are: (i) delta-record format, flexible configuration of IPA and database page layout; (ii) DBMS operations - fetching, modification and eviction; (iii) error-correction codes (ECC) on Flash; (iv) program interferences on Flash.

Delta-record, N×M scheme and database page layout.

Delta-records store information needed to reconstruct the up-to-date version of the page. Updates are "logged" in byte-granularity,

[1] The IPL versus IPA comparison was done by using the original IPL simulator and the Flash memory configuration from [8] on traces recorded from running TPC-B/-C and TATP benchmarks.

[2] On SLC Flash this means that all updated bits change from 1 to 0

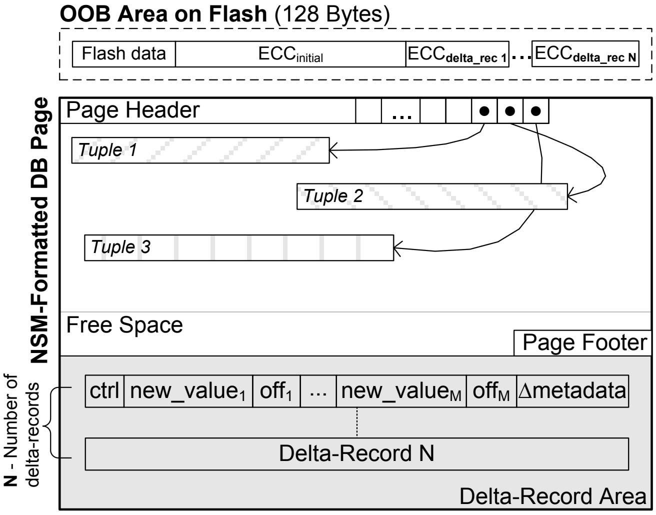


Figure 3: Database page-format, supporting IPA on Flash

i.e. each updated byte is represented in delta-record as a <*new_value, offset*> pair. The configuration parameter M determines the maximum number of such pairs stored in a single delta-record. Furthermore, each delta-record contains: (i) a *control_byte* - a flag representing the presence of the delta-record, and (ii) the modified version of page metadata: header and footer (see Figure 3). To "accommodate" the delta-records on the page we reserve a certain amount of space at the end of the page – the so-called delta-record area. The number of delta-records per page is controlled by the configuration parameter N. Thus, the delta-record area size for a particular $N{\times}M$ configuration is: $N \times (1 + 3M + \Delta metadata)$.

Page operations. IPA requires certain modifications in the traditional operations on database pages. Before the page is placed into the buffer frame upon being fetched, the storage manager checks if it contains delta-records. If so, those are applied by changing the original bytes at defined offsets to their updated values from the delta-records. Now the page body is in its up-to-date state. Similarly, the page metadata is updated to its actual version from $\Delta metadata$ in the delta-record. Finally, the resulting page is placed into the buffer frame.

When a transaction updates the content of the page, the buffer manager checks if it conforms to the IPA $N{\times}M$ scheme. Thus, the total number of delta-records (including the existing) cannot exceed N, while the number of changed bytes per delta-record should not exceed M. If those conditions are fulfilled, the update is performed as usual, while the offsets of changed bytes are stored in the delta-record(s). The traditional behavior of the buffer manager is not affected by IPA, since the buffer contains always the up-to-date version of the page, and all updates are done as usually in-place. The violation of one of the above conditions means that upon eviction the page cannot be written out using IPA, and will therefore be written in a traditional out-of-place manner on Flash. In this case, the *out-of-place flag* is set, and further updates are not tracked until eviction.

On page eviction from the buffer pool, the storage manager checks whether the *out-of-place flag* is set. If so, the delta-record area is reset, and the up-to-date version of the page is written out in an out-of-place manner. Otherwise, the page can be overwritten in-place by using in-place appends. In this case, only the delta-record(s) is transmitted to the Flash storage by using the *write_delta()* command.

write_delta(LBA, offset, delta_length, delta_bytes[ ]);

The delta-record(s) will be appended to the very same physical Flash page containing the original database page. This is possible since the original content of the page is left unchanged, while all updates are coalesced in the appended delta-record. *The sole transfer of delta-records (instead of whole pages) significantly reduces the DBMS write-amplification, whereas appending those delta-records to original Flash pages eliminates the need to perform page invalidations and out-of-place writes, which further reduces the GC overhead (on-device write-amplification).* IPA is also applicable to conventional SSDs with block-device interface (see Section 4).

Please note that the regular database functionality (e.g. recovery, locking, etc.) is NOT impacted by the proposed approach. Furthermore, it introduces negligible or no overhead to the DBMS, since (i) it can be selectively applied only to specific database objects using *NoFTL Regions*; (ii) change tracking in the buffer produces min. computational overhead.

Flash types and program interference. In-Place Appends can be applied to all modern types of Flash memory, namely SLC, MLC/eMLC and TLC in 3D NAND. On **SLC** NAND Flash IPA can be applied without specific limitations. The reason is that the difference between different threshold voltages (indicating different logical bit-codes of the Flash cell: 1 and 0) is large enough to compensate small deviations which might appear due to program interference (parasite capacitance-coupling), while (re-)programming the Flash-page (appending the delta-record). The **MLC** Flash is more susceptible to the program interference errors, due to the shorter distances between different voltage thresholds. To safely apply In-place Appends on MLC Flash without increasing program interference we propose two configuration modes. First, the MLC Flash can be used in *pseudo-SLC mode (pSLC)*: the Flash capacity halved as very second page of Flash memory is effectively used (LSB-pages). In this mode the MLC Flash is as tolerant to program interference errors as SLC Flash. Under the second, also called *odd-MLC* mode, the whole MLC Flash capacity is utilized. However, IPA are only applied to LSB pages (odd numbered pages), whereas MSB pages (even numbered pages) still need to be programmed in standard out-of-place manner. **3D NAND** Flash addresses program interference issues by using new manufacturing technologies. According to Samsung their 3D V-NAND chips are: "Bitline Interference Free" and "Wordline Interference Almost Free" [2]. Therefore, IPA is applicable to 3D NAND using the above SLC/pSLC or odd-MLC techniques.

4. DEMONSTRATION

During the demonstration we introduce the audience to basics of the proposed approach and let them evaluate it interactively on real hardware. The demonstration system consists of the Flash storage - the OpenSSD research Flash board[3] connected to a host PC running Shore-MT storage engine (Figure 4). Using an intuitive GUI (Figure 5) the audience can configure a sequence of tests and experience live the performance advantages of the IPA. The proposed demonstration scenarios are as follows.

Demo-Scenario 1 – Baseline.

The audience picks one of the three available OLTP benchmarks

[3] Four dual-die Samsung K9LCG08U1M 8GB packages per module. Each package consists of 4096 erase units each holding 128 16KB Flash pages [1].

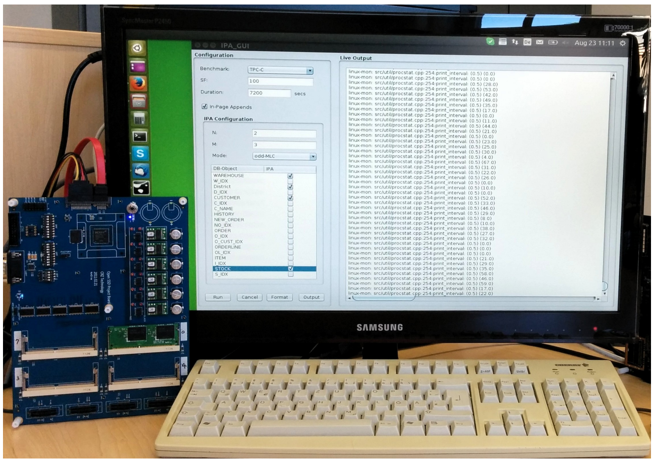
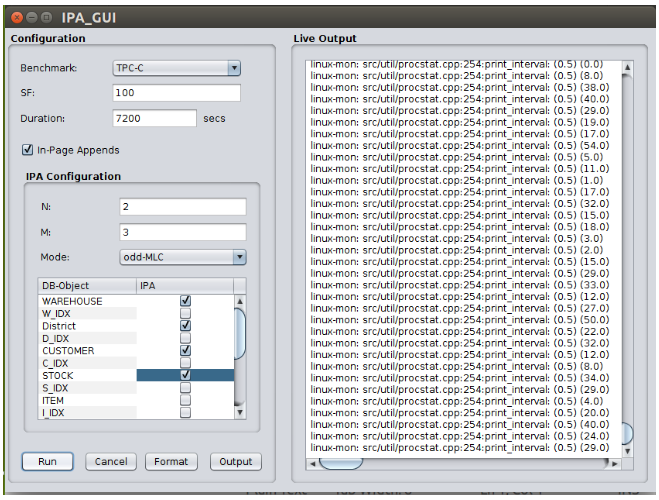


Figure 4: Demonstration system with the OpenSSD board.



Benchmark: TPC-C

SF: 100

Duration: 7200 secs

☑ In-Page Appends

IPA Configuration

N: 2

M: 3

Mode: odd-MLC

DB-Object	IPA
WAREHOUSE	☑
W_IDX	□
District	☑
D_IDX	□
CUSTOMER	☑
C_IDX	□
STOCK	☑
S_IDX	□
ITEM	□
I_IDX	□

```
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (0.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (8.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (38.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (40.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (29.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (19.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (17.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (54.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (5.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (11.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (1.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (17.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (32.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (15.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (18.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (3.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (2.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (15.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (29.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (33.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (12.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (27.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (50.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (22.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (32.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (12.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (8.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (34.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (29.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (4.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (20.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (40.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (24.0)
linux-mon: src/util/procstat.cpp:254:print_interval: (0.5) (29.0)
```

Figure 5: GUI for the evaluation of IPA.

(TPC-B, TPC-C or TATP), selects the desired scaling factor (limited by 64GB of Flash storage) and the duration of the test. The DBMS executes the benchmark using the traditional approach as a baseline, i.e. every updated DB-page results in one or more out-of-place writes on Flash. During the benchmark run the audience can observe the current transactional throughput. At the end detailed statics of performed I/Os are visualized.

Demo-Scenario 2 – IPA for conventional SSD.

In this scenario the audience examines IPA designed for conventional SSDs. Its implementation assumes the use of traditional block-device interface. The DBMS writes out whole pages in the format: page body + delta-record area. After the main parameters of IPA have been selected ($N\times M$ scheme and the mode of IPA on MLC Flash: pSLC or odd-MLC), and the Flash SSD is completely formatted (low-level formatting) the benchmark is run with the same scaling factor and for the same duration as in the baseline test. The audience can compare the output results of both approaches (throughput, I/O statistics).

Demo-Scenario 3 – IPA for native Flash.

This scenario is similar to the previous one, however, the DBMS utilizes IPA designed for native Flash (e.g. NoFTL architecture). In this case only the delta-records are transferred to the Flash storage. Both IPA scenarios #2 and #3 result in the same reduction of GC overhead, since in both cases updates are performed as in-place appends reducing the number of page invalidations. However, here IPA uses *write_delta* command, which significantly reduces the DBMS write-amplification and the amount of transferred data.

Table 1: TPC-B: traditional approach (no In-Place Appends [0×0]) vs. [2×4] scheme in modes pSLC and odd-MLC.

	0x0	2x4	2x4	2x4	2x4
	Absolute	Absolute	Relative	Absolute	Relative
		pSLC	*pSLC* [%]	*odd-MLC*	*odd-MLC* [%]
Out-of-Place Writes vs. In-Place Appends			33/67		51/49
Host Reads (16KB)	3 779 926	5 540 034	+47	4 875 961	+29
Host Writes (16KB)	2 028 626	3 047 538	+50	2 372 017	+17
GC Page Migrations	605 047	153 201	-75	315 228	-48
GC Erases	15 839	7 401	-53	7 625	-52
Page Migrations per Host Write	0.2983	0.0503	-83	0.1329	-55
GC Erases per Host Write	0.0078	0.0024	-69	0.0032	-59
Transactional Throughput	260	380	+46	313	+20

Table 1 shows the comparison results of TPC-B benchmark running for two hours on OpenSSD board (during the demonstration the durations of 5 or 10 minutes are sufficient for a comparison). The experiments were performed (i) without IPA ([0×0] column), and with IPA using (ii) pSLC and (iii) odd-MLC modes with [2×4] configuration scheme. Under TPC-B, IPA outperforms the traditional approach by executing up to 70% less erases and up to 85% less page migrations. This reduction of GC overhead has two major advantages: (i) the increase of the transactional throughput of up to 45%, and (ii) doubling the Flash SSD lifetime.

Acknowledgments

This paper was supported by the German BMBF "Software Campus" (01IS12054) and the German Research Foundation (DFG) project "Flashy-DB".

5. REFERENCES

[1] The openssd project. http://www.openssd-project.org, 2014.

[2] Samsung v-nand. http://www.samsung.com/us/business/oem-solutions/pdfs/V-NAND_technology_WP.pdf, 2014.

[3] S. Aritome. *NAND flash memory technologies*. IEEE Press series on microelectronic systems. Wiley-IEEE Press, 2016.

[4] 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*.

[5] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann. From in-place updates to in-place appends: Revisiting out-of-place updates on flash. In *Proc. SIGMOD'17*.

[6] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann. Noftl: Database systems on ftl-less flash storage. In *Proc. VLDB'13*.

[7] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann. Revisiting dbms space management for native flash. In *Proc. EDBT*, 2016.

[8] S.-W. Lee and B. Moon. Design of flash-based dbms: An in-page logging approach. In *Proc. SIGMOD'07*.

[9] Y. Lu, J. Shu, and W. Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proc. FAST'13*.

CAESAR: Context-Aware Event Stream Analytics for Urban Transportation Services

Olga Poppe*, Chuan Lei**, Elke A. Rundensteiner*, Dan Dougherty*,
Goutham Deva*, Nicholas Fajardo*, James Owens*, Thomas Schweich*,
MaryAnn VanValkenburg*, Sarun Paisarnsrisomsuk*, Pitchaya Wiratchotisatian*,
George Gettel*, Robert Hollinger*, Devin Roberts*, and Daniel Tocco*
*Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609
**NEC Labs America, 10080 N Wolfe Rd, Cupertino, CA 95014
*opoppe|rundenst|dd|godeva|nafajardo|jmowens|taschweich|mevanvalkenburg|
spaisarnsrisomsu|pwiratchotisatia|gtgettel|rhollinger|dtroberts|dtocco@wpi.edu,
**chuan@nec-labs.com

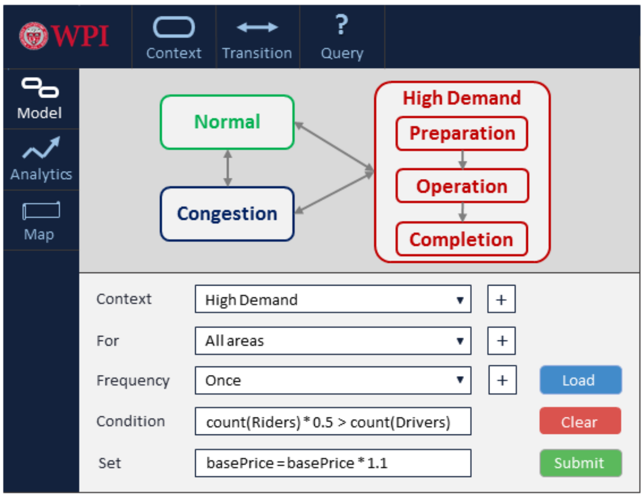
ABSTRACT

We demonstrate the first full-fledged context-aware event processing solution, called CAESAR[1], that supports application contexts as first class citizens. CAESAR offers human-readable specification of context-aware application semantics composed of context derivation and context processing. Both classes of queries are only relevant during their respective contexts. They are suspended otherwise to save resources and to speed up the system responsiveness to the current situation. Furthermore, we demonstrate the context-driven optimization techniques including context window push-down and query workload sharing among overlapping context windows. We illustrate the usability and performance gain of our CAESAR system by a use case scenario for urban transportation services using real data sets [2, 1].

1. INTRODUCTION

Context-Aware Event Stream Analytics. Complex Event Processing (CEP) is a prominent technology for supporting time-critical applications. Traditionally, CEP systems consume an event stream and continuously evaluate the *same* query workload against the *entire* event stream. However, the semantics of many streaming applications is determined by contexts, meaning that *the system reaction to one and the same event may significantly vary depending on the context*. Therefore, most event queries are appropriate only under certain circumstances and can be safely suspended otherwise to save valuable resources and reduce the latency of currently relevant queries.

Running Demonstration Scenario. With the growing popularity of Uber and Lyft, their real-time systems face a wide range of challenges, including but not limited to extracting supply/demand sequence patterns from event streams, real-time aggregation, geospatial prediction, traffic data monitoring and alerting. These data intensive event queries continuously track the status of drivers, riders, and traffic, such as driver dispatched, rider waiting for pickup, road congestion, etc. An intelligent event processing system receives both vehicle and rider position reports and their associated messages, analyzes them, infers the current supply and demand situation in each geolocation, and reacts instantaneously to ensure that riders reach their destinations in a timely and cost-effective manner. Early detection and prompt reaction to critical situations are indispensable. They prevent time waste, reduce costs, increase riders' satisfaction and drivers' profit.



Context: High Demand
For: All areas
Frequency: Once
Condition: count(Riders) * 0.5 > count(Drivers)
Set: basePrice = basePrice * 1.1

Figure 1: The CAESAR model

System reaction to a position report should be modulated depending on the *current situation on the road*. Indeed, if *HighDemand* is detected, all drivers close by are notified and a higher fee is charged in this area to attract more drivers and reduce the waiting time of riders (Figure 1). If a road segment becomes *Congested*, drivers may be alerted and alternate routes should be advised so as to smooth traffic flow. If a road segment is *Normal*, none of the above

[1] CAESAR stands for Context-Aware Event Stream Analytics in Real time.

Series ISSN: 2367-2005 10.5441/002/edbt.2017.77

actions should take place. Clearly, *current application contexts* must be rapidly detected and continuously maintained to determine appropriate reactions of the system at all times.

The hierarchical application logic in Figure 1 is drilled down into the *HighDemand* specification, while the interior structures of all other processes are rolled up and thus abstracted to increase readability. Three contextual stages are differentiated during the *HighDemand* context. First, *all* drivers in proximity are notified and a higher base fee is computed (*Preparation*). Afterwards, only *new* nearby drivers are notified and the high base fee is used to compute the cost of each trip (*Operation*). Lastly, the base fee is reduced once demand is satisfied (*Completion*). Appropriate event queries are associated with each context. For example, new drivers are detected during the *Operation* phase in a high demand geolocation (Figure 1).

Conditions implying an *application context* can be complex. They are specified on both the event streams and the current contexts. For example, if over 50 cars per minute move with an average speed less then 40 mph and the current context is no *Congestion* then the *context-deriving query* updates the context to *Congestion* for this geolocation. To save resources and thus to ensure prompt system responsiveness, such complex context detection should happen once. Its results must be available immediately and shared among all queries that belong to the detected context. In other words, *context-processing* queries are *dependent* on the results of *context-deriving* queries. A synchronization mechanism ensuring their correct execution must be employed.

Challenges. To enable real-time responsiveness of such applications, the following challenges must be tackled.

Context-aware specification model. Many streaming applications have context-driven semantics. Thus, they must support application contexts as first class citizens and enable linkage of the appropriate event query workloads to their respective contexts in a modular format to facilitate on-the-fly reconfiguration, easy maintenance, and avoid fatal specification mistakes.

Context-exploiting optimization techniques. To meet the demanding latency constraints of time-critical applications, this context-aware application model must be translated into an efficient physical query plan. This is complicated by the fact that the duration of a context is unknown at compile time and potentially unbounded. Furthermore, contexts are implied by complex conditions. They are interdependent and may overlap.

Context-driven execution infrastructure. An efficient runtime execution infrastructure is required to support multiple concurrent contexts. To ensure correct query execution, the inter-dependencies between context-deriving and context-processing queries must be managed effectively.

State-of-the-Art Approaches. Traditional CEP windows fail to express variable-length inter-dependent context windows. Indeed, tumbling and sliding windows [8] have fixed length, while predicate windows [6] are defined independently from each other.

Most graphical models express either only the workflow [4] or only single event queries [3]. Some models and event languages can express contexts by procedures [7] or queries [5]. However, they do not allow for the modular specification of context-driven applications – placing an unnecessary burden on the designer [9, 10]. Furthermore, optimization techniques enabled by contexts are yet to be developed.

Contributions. We demonstrate the following contributions of the CAESAR technology: (1) The easy-to-use graphical interface to illustrate the powerful CAESAR model [9, 10]. It visually captures application contexts, transitions between them, context-deriving and context-processing queries. (2) The optimization techniques enabled by contexts suspend those queries that are irrelevant to the current context and share computations between overlapping contexts. (3) The CAESAR infrastructure guarantees correct and efficient context management at runtime. (4) We illustrate the usability and performance gain of the CAESAR technology using the real-world urban transportation scenario [2, 1].

2. CAESAR SYSTEM OVERVIEW

Figure 2 provides an overview of the CAESAR system.

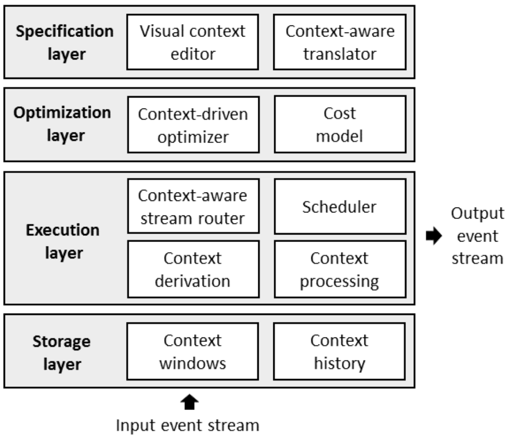


Figure 2: The CAESAR system

Specification Layer. The designer specifies the CAESAR model (Section 3.1) using the visual context editor. The model is then translated it into an algebraic query plan.

Optimization Layer. The query plan is optimized using the context-driven optimization techniques (Sections 3.2 and 3.3) to produce an efficient execution plan.

Execution Layer. The optimized query plan is forwarded to the scheduler that guarantees correct context derivation and processing at runtime (Section 3.4).

Storage Layer. Context windows and history are compactly stored and efficiently maintained at runtime.

3. KEY INNOVATIONS OF CAESAR

3.1 Context-aware Event Query Model

While the CAESAR model is formally defined in [9, 10], below we briefly summarize its key components and benefits.

Application Contexts are real-world higher-order situations the duration of which is not known at their detection time and potentially unbounded. This differentiates contexts from events. The duration of an application context is called a *context window.* For example, *Congestion* is a higher-order situation in the traffic use case. Its bounds are detected based on position reports of cars in the same area at the same time. As long as a road remains congested, the context window *Congestion* is said to hold. Hence, the duration of a context window cannot be predetermined.

At each point of time, the CAESAR model re-targets all efforts to the current situation by activating only those context-deriving and context-processing queries which handle the current contexts. Irrelevant queries are suspended to save resources. For example, Uber surge pricing kicks in only during *HighDemand* on a road. This query is neither relevant in the *Normal* nor in the *Congestion* contexts. Thus, it is evaluated only during *HighDemand* and suspended in all other contexts.

Context-Deriving Queries are associated with a particular context and determine when this context is terminated and when a particular other context is initiated based on events. For example, once many slow cars on a road are detected during the *Normal* context the system transitions into the *Congestion* context. Thereafter, the query detecting *Congestion* is no longer evaluated. All event queries that are evaluated during *Congestion* leverage the insight detected by the context-deriving query rather than re-evaluating the *Congestion* condition at each individual query level.

Context-Processing Queries react to events that arrive during a context in an appropriate way. Contexts provide queries with *situational knowledge* that allows to specify simpler event queries. For example, if the query computing surge pricing is evaluated only during the *HighDemand* context, the complex conditions that determine that there is a high demand in this geolocation are already implied by the context. Thus, there is no need to repeatedly double-check them in each of the context-processing queries.

3.2 Context Window Push-Down Optimization

Our CAESAR algebra consists for the following six operators: Context initiation, context termination, context window, filter, projection, and pattern [10]. With filter, projection and pattern common in stream algebras [12], traditional multi-query optimization techniques [11] are applicable to our CAESAR queries. In addition, we propose two context-driven optimization techniques, namely we push context windows down and share workloads of overlapping contexts. Pushing context windows down in a query plan prevents the continuous execution of operators "out" of their respective contexts and thus reduces the costs. To guarantee correctness, we group event queries by contexts. By definition, a context window specifies the scope of its queries. Thus, pushing a context window down in each group of queries does not change the semantics of these queries.

In contrast to traditional predicates, context windows are not just filters on a stream that select certain events to be passed through. Context windows *suspend* the entire query plan "above them" as long as the application is in different contexts. Furthermore, our context-driven stream router directs entire *stream portions during contexts* to their respective queries (Section 3.4) rather than filtering events one by one at the *individual event level* which is a resource-consuming process.

3.3 Context Workload Sharing Optimization

Similar computations may be valid in different contexts. For example, an accident on a road is detected during all contexts in Figure 1. In such cases, substantial computational savings can be achieved by sharing workloads between overlapping contexts. For example, *Congestion* and *HighDemand* may overlap. To avoid repeated computations and storage, we split the original user-defined *overlapping* context windows into finer granularity context windows and group them into *non-overlapping* context windows by merging their workloads. Within each newly produced non-overlapping context window, we apply traditional multi-query optimization techniques [11]. Our context window grouping strategy divides the query workload into smaller groups based on their time overlap. As additional benefit, the search space for an optimal query plan within each group is substantially reduced compared to the global space.

3.4 CAESAR Execution Fabric

The core of the CAESAR execution fabric consists of the context derivation, context-aware stream routing, context processing, and scheduling of these processes (Figure 2). While we briefly describe these components below, we refer an interested reader to our full paper [10] for more details.

Context Derivation. For each stream partition (a geolocation in the traffic use case), the context bit vector W maintains the currently active contexts. This vector W has a time stamp *W.time* and a one-bit entry for each context. The entry 1 (0) for a context c means that the context c holds (does not hold) at the time *W.time*. Since contexts may overlap, multiple entries in the vector may be set to 1. *W.time* is the application time when the vector W was last updated by the context-deriving queries. This time stamp is crucial to guarantee correctness of interdependent queries.

Context-Aware Stream Routing. Based on the context bit vector, the system is aware of the currently active contexts. For each current context c, the system routes all its events to the query plan associated with the context c. Query plans of all currently inactive contexts do not receive any input. They are suspended to avoid waste of resources.

Context Processing. The CAESAR model uses contexts to specify the *scope* of queries. When a user-defined context ends, all associated queries are suspended and thus will not produce new results until they become activated again. Hence, their partial results, called *Context history*, can be safely discarded. However, if a user-defined context c with its associated query workload Q^c is split into smaller non-overlapping contexts c_1 and c_2, then partial results of the queries Q^c must be maintained across these new contexts c_1 and c_2 to ensure completeness of the queries Q^c.

Correctness. Context-processing queries are dependent on the results of context-deriving queries. To avoid race conditions and ensure correctness, these inter-dependencies must be taken into account. To this end, we define a *stream transaction* as a sequence of operations that are triggered by all input events with the same time stamp. An algorithm for scheduling read and write operations on the shared context data is *correct* if conflicting operations[2] are processed by sorted time stamps. While existing stream transaction schedulers could be deployed in the CAESAR system, we currently deploy a time-driven scheduler.

4. DEMONSTRATION SCENARIO

In this section, we demonstrate the above key innovations of the CAESAR system based on the urban transportation services using two real data sets [2, 1] that contain millions of taxi and Uber trips in New York city in 2014 and 2015.

Visual CAESAR Model Design. The audience will view and edit CAESAR models using simple drag-and-drop interaction tools. Figure 1 shows that the model captures

[2] Two operations on the same value such that at least one of them is a write are called conflicting operations.

the complex application logic in a succinct and readable manner. The audience can view the specification at different levels of abstraction. There are three composed contexts, namely, *Normal, Congestion*, and *High Demand*. All other contexts are atomic. The composed contexts can be collapsed and expanded with a click of a button. For ease of follow-through, color schemas of composed contexts and their interior structures are consistent. To keep the model clean and readable, the contexts and transitions between them are depicted in the middle panel separately from their respective context-deriving and context-processing queries shown in the bottom panel. When the cursor is over a transition, its corresponding context-deriving query appears as a label of the transition. When the designer clicks on a label, (s)he can conveniently edit it in the bottom panel. Similarly, when the designer clicks on a context, the list of context-processing queries appears in the bottom panel. We will demonstrate the ease with which CAESAR models can be dynamically reconfigured by editing contexts, transitions between them, and their respective queries.

Execution Visualization. At runtime, the model view provides insights into event-driven context transitions (Figure 1). The current context and triggering transitions are temporally highlighted. Besides the real-time monitoring, the model view offers a slow-motion-replay mode that allows the users to step-through the history of prior execution to better understand, debug, and reconfigure the model. This functionality provides the audience a visual opportunity to learn how the CAESAR model functions.

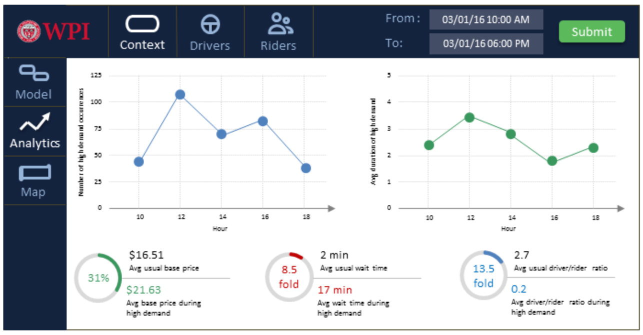


Figure 3: Analytics view

Execution Optimization. The analytics view will allow the audience to monitor the effect of the context-driven optimization techniques. The audience will first chose to show statistics either about contexts, or drivers, or riders in the top panel of Figure 3. Also, the audience can specify the time interval of interest in the top panel. Thereafter, charts visualizing runtime statistics will appear in the middle panel. They provide a summary about the chosen topic during the time interval of interest. For example, to summarize the contextual information, the number of high demand occurrences, average duration of this context, as well as the price, wait time, and driver vs. rider ratio during 8 hours are compactly presented in Figure 3.

Interactive City Map offers the audience an abstract view of the current situation by highlighting the areas in different colors depending on their contexts. For example, a high demand area is identified in the middle panel in Figure 4 highlighted by a red circle. Green and blue circles visualize riders outside of the high demand area. In addition to the map, runtime statistics are shown in the top panel. They include the number of current high demand or congested areas, the number of recent requests and current trips, the number of available drives and waiting riders.

In addition to the complex events that are automatically derived by queries, the audience will learn about common manual actions which include area specific information such as accidents, road construction, gas prices, police cars etc. This information will be added by clicking on the respective location on the map and choosing the information in a drop-down menu. A respective icon will appear on the map. For example, one traffic hazard is depicted in Figure 4. Based on this information, travel time and cost will be estimated to compute the best route of each trip.

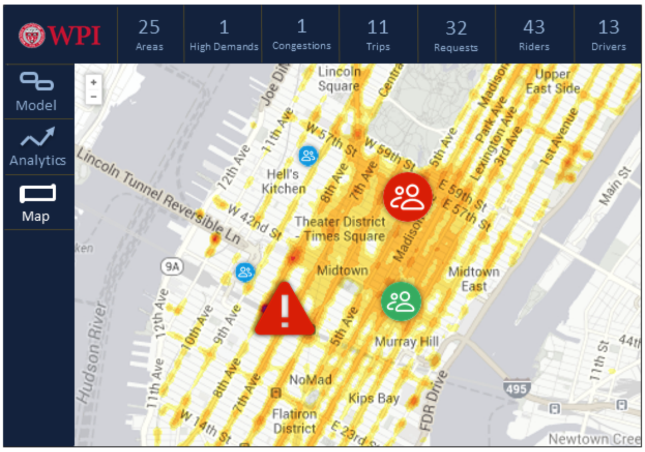


Figure 4: Map view

Conclusion. The CAESAR technology offers a principled end-to-end solution for context-aware stream analytics.

Acknowledgement. This work was supported by NSF grants IIS 1018443, IIS 1343620, IIS 1560229, and CRI 1305258.

5. REFERENCES

[1] Uber TLC FOIL Response. https://github.com/fivethirtyeight/uber-tlc-foil-response.

[2] Unified New York City Taxi and Uber data. https://github.com/toddwschneider/nyc-taxi-data.

[3] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, 2008.

[4] R. Alur and D. Dill. Automata for modeling real-time systems. In *ICALP*, pages 322–335, 1990.

[5] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB Journal*, 15(2):121–142, 2006.

[6] T. M. Ghanem, W. G. Aref, and A. K. Elmagarmid. Exploiting predicate-window semantics over data streams. *SIGMOD Rec.*, 35(1):3–8, 2006.

[7] A. Grosskopf, G. Decker, and M. Weske. *The Process: Business Process Modeling using BPMN*. Meghan Kiffer Press, 2009.

[8] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. Tucker. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. In *SIGMOD*, pages 39–44, 2005.

[9] O. Poppe, S. Giessl, E. A. Rundensteiner, and F. Bry. The HIT model: Workflow-aware event stream monitoring. In *TLDKS*, pages 26–50. 2013.

[10] O. Poppe, C. Lei, E. Rundensteiner, and D. Dougherty. Context-aware event stream analytics. In *EDBT*, pages 413–424, 2016.

[11] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, pages 249–260, 2000.

[12] E. Wu, Y. Diao, and S. Rizvi. High-performance Complex Event Processing over streams. In *SIGMOD*, pages 407–418, 2006.

Building Multi-Resolution Event-Enriched Maps From Social Data

Faizan Ur Rehman[*]
Science and Technology Unit,
Umm Al-Qura University, KSA
fsrehman@uqu.edu.sa

Imad Afyouni
Technology Innovation Center
Wadi Makkah, KSA
iafyouni@gistic.org

Ahmed Lbath
LIG, University of Grenoble
Alpes, France
Ahmed.Lbath@imag.fr

Sohaib Ahmad Khan[†]
Science and Technology Unit
Umm Al-Qura University, KSA
skhan@gistic.org

Saleh Basalamah
College of Computer and
Information Systems, Umm
Al-Qura University, KSA
smbasalamah@uqu.edu.sa

Mohamed Mokbel
Department of Computer
Science and Engineering,
University of Minnesota, USA
mokbel@cs.umn.edu

ABSTRACT

This paper discusses the next generation of digital maps, by positing that maps in future will intelligently self-update themselves based on distinctive events extracted dynamically from social media streams or other crowd-sourced data. To realize this concept, the challenges include developing a scalable and efficient system to deal with a variety of unstructured data streams, applying NLP and clustering techniques to extract relevant information from these streams, and inferring the spatio-temporal scope of detected events. This paper demonstrates *Hadath*, a system that extracts live events from social data by encapsulating incoming unstructured data into generic *data packets*. The system implements a hierarchical in-memory indexing scheme to support efficient access to data packets, as well as for memory flushing purposes. Data packets are then processed to extract *Events of Interest* (EoI), based on a multi-dimensional clustering technique. Next, we establish the spatial scope and the level of abstraction of each event. This allows us to show live events in correspondence to the scale of the view – when viewing at a city scale, we see events of higher significance, while zooming in to a neighborhood highlights events of a more local interest. The final output creates a unique and dynamic map browsing experience.

CCS Concepts

•**Information systems** → **Geographic information systems; Wrappers (data mining); Data streaming;**

[*] Also affiliated with LIG, University of Grenoble Alpes, France

[†] Also affiliated with Department of Computer Science, Lahore University of Management Sciences, Lahore, Pakistan

Keywords

Event-Enriched Maps; Crowdsourced Data; Spatio-Temporal Scope

1. INTRODUCTION

While it is the norm nowadays to use digital mapping applications to use live data to find directions, traffic congestion states or places of interest, we posit that the next generation of maps will contain the additional functionality of showing live events at different spatio-temporal resolutions, and which are extracted dynamically from a variety of sources, starting from online social media and crowd-sourced data, to open governmental data and other online news sources. Within this context, there is a real opportunity to enrich current maps with knowledge extraction tools that take advantage of information retrieval, data management, and sentiment analysis techniques. Analyzing crowdsourced data can provide deep insights about surrounding events of interest (EoI). For instance, with the explosive growth in size of microblog data (e.g., Twitter, Flickr, and Yelp), fruitful insights can be extracted and displayed (examples of discovered findings are illustrated in Figure 1). However, designing an efficient and scalable system that extracts live events and infers their spatial and temporal scopes, so that they can be displayed in a clear, non-cluttered manner, remains a challenging task.

The challenge of displaying live events on a map is three-fold. Firstly, these events need to be extracted from unstructured data streams efficiently, while preserving accuracy and conciseness. Secondly, to display such events on a map, their spatial scope must be established, so that as a user changes the zoom level, only events of appropriate scope are displayed. For example, a soccer match may be displayed at the city scale, the opening of a new restaurant at sub-urban scale, and a house-warming party at the neighborhood scale. Thus, not only is it necessary to extract the events themselves, but also to establish their spatial scope, so that they can be displayed appropriately in a clutter-free manner. Finally, all of this has to be done in real-time so that live streams can be handled, and up-to-date events at multiple resolutions can be detected. To address these challenges, we propose *Hadath*, a system that han-

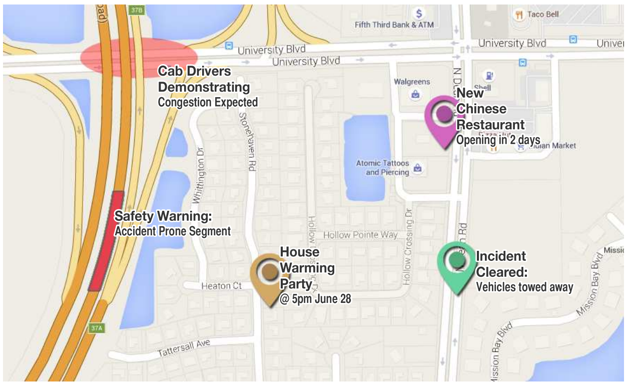


Figure 1: Conceptual illustration of Event-Enriched Maps: Findings automatically discovered from live streams, such as a restaurant opening, a neighborhood party, an accident prone road segment, warnings on demonstrations and emergency cases.

dles unstructured social streams, particularly Twitter data, and implements different algorithms for the efficient extraction, clustering, and mapping of live crowdsourced events. *Hadath* consists of several components as follows. Data collection involves gathering social data with different forms: data chunks and streams. The data wrapping and cleaning component digests streaming data, and prepossesses data to generate structured data packets from unstructured streams. The data manager stores data by implementing a indexing scheme to allow efficient and scalable access to raw data, as well as to extracted events. The events of interest detection module classifies and extracts events based on a multidimensional and hierarchical clustering technique, which defines the spatial scope and the level of abstraction of detected events. The query engine creates best query plan based on map zoom level, spatial and temporal characteristics, and executes the query plan in order to retrieve EOIs efficiently. The visualizer provides a new dimension to existing maps by illustrating extracted knowledge from live collected data as live events at different levels of abstraction. The remainder of this paper is as follows. Section 2 highlights related work and challenges from different perspectives. Section 3 introduces our proposed architecture with results; while Section 4 draws conclusions and discusses future work.

2. RELATED WORK

This section highlights different challenges and state-of-the-art techniques related to: 1) digital mapping, 2) events of interest detection, and 3) performance and scalability perspectives.

1. State-of-the-art mapping technologies: Today's maps are often crowd-sourced, and make use of *'Volunteered Geographic Information (VGI)'*, where users can seed maps with their own content. Researchers, authorities, and industries generate thousands of map-based analytics every year to meet their social and economic needs [6]. In addition, 'Live Maps' now contain real-time updates of bus schedules, traffic conditions, restaurant opening hours, and road accidents, among others. With the wide spread of social networks, people start to post their own social contributions on live maps, such as Foursquare check-ins, Flickr images, tweets [7], and Yelp reviews. Moreover, NLP techniques were embedded to extract spatially-referenced news from online newspapers and tweets [10]. However, current maps still lack intelligence in extracting knowledge about new events occurring at different spatio-temporal resolutions.

2. Discovering Events of Interest along with Spatio-Temporal Scope: Detection of irregular happenings and trends from social data, mainly Twitter data, is already a topic of many scientific articles [2]. This mainly includes: 1) earthquake detection along with their centers or other natural disasters; 2) extracting and localizing breaking news from tweets as presented in TwitterStand [10]; 3) discovering incidences related to traffic conditions [8] from twitter and user generated data; and 4) detection of unspecified hot topics based on text similarity [1], density or with wavelet spatial analysis [5]. The work presented in [4] is very close to our work with respect to detecting spatial/temporal extents of events, but was only distinguishing between local and global scales, without putting focus on mapping those events to the different spatio-temporal resolutions in digital maps.

3. Performance and Scalability Perspectives: Several works have presented systems that visualize geo-tagged social streams on maps, such as Flickr images[1], tweets [7], Yelp reviews, and spatially-referenced news [10]. Particularly, NewsStand [10] is a scalable system that extracts news from RSS feeds and visualize them on a world wide map. Furthermore, the system can apply spatio-temporal and keyword-based filtering of news. However, this system displays news at different spatial scales by only ranking them based on the number of views, without detecting and clustering events of interest along with their spatial scope. With the large volume of incoming streams, data indexing and the distributed processing of data represent an essential part of any system that implements *'event-enriched maps'*.

3. SYSTEM OVERVIEW

This section presents *Hadath*, a system that retrieves data streams from social data (here we focus on Twitter data), efficiently manages and processes those streams in order to find Events of Interest (EoI), and visualizes those events in correspondence to their spatial and temporal scopes, thus creating *'multi-resolution event-enriched maps'*. Figure 2 illustrates the main components of our system architecture, which are described as follows:

- *Data collection* involves gathering data with different format. This includes digesting data streams and data chunks (i.e., historical tweets) from Twitter. In data chunks mode, we download the files that contain partial or full datasets. Digesting data streams is performed by running crawlers that collects bulks of streams based on windows of a specified temporal extent w (e.g., 30 minutes window).
- The *data wrapper* provides an efficient and generic mechanism with the aim of allowing new data sources (e.g., Flickr) to be easily plugged, by supporting new crawlers at the data collection level without affecting the other processing components. Major tasks for the data wrapper are: 1) to clean irrelevant fields and digest incoming streams into a unique data packet format; 2) to use specified string matching technique that detect and match candidate packets with our event classifier corpus in order to identify potential event

[1] https://www.flickr.com/map

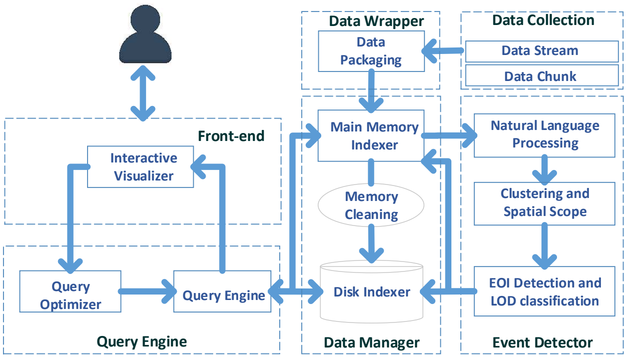
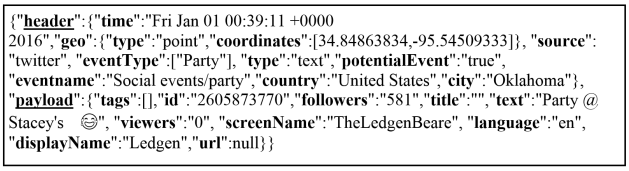


Figure 2: Hadath Architecture

classes and properties; and 3) to apply unspecified topic detection method that extract spatio-temporal peaks and unusual happenings based on the top frequent words. Figure 3 shows an example of data packets generated from twitter data with potential event flag, event class name, and event properties. Packets that show no relevancy with respect to the above steps are discarded at this phase.

• The *data manager* implements an in-memory spatial indexing scheme to allow efficient and scalable access to data packets. The spatial index is a multi-resolution data structure (similar to a partial quad tree [9]). Leaves in this data structure correspond to cells that represent the minimum bounding rectangles comprising data packets. Figure 4 displays a snapshot of indexed data packets at a fine level of the hierarchal tree, and with a single day specified as a time threshold. Cells are colored lighter to darker based on data packet counts; darker-colored cells are further expanded at deeper levels in the tree as compared to lighter-colored cells. *Hadath* employs a big data mechanism that continuously process data packets within the different cells on several execution nodes. The manager also indexes detected EoIs in order to fetch them efficiently based on the map zoom level and scope. Using this multi-resolution indexing scheme, hierarchical clustering of events can be applied for efficient determination of their content and spatial scopes. For temporal aspects and cleaning of EoIs, we took three parameters: a) *'birth time'* that indicates the existence of a new event in our system whenever we calculate the first cluster of data packets related to that event; b) *'time of occurrence'* that marks the actual happening time of the event (e.g., next Monday); and c) *'time to live'* (TTL) is the survival time of an event in our system. Whenever we receive new data packets related to an existing event, we increase its TTL by T number of hours. Moreover, processed data packets are moved to disk based on temporal and memory thresholds. The main task of disk indexer is to index outdated data packets and events on disk using an R*-tree spatial index to allow efficient retrieval for historical queries.

• The *event detector* module starts from leaf cells within the multi-resolution data structure to detect events at a local spatio-temporal scope. Within each leaf cell, our system adopts the graph analogy where each potential event data packet is considered as a *node* and the value of 'text similarity (TF-IDF)' between data packets as a weight of the bidirectional *edge*. Data packets with a high text similarity value are clustered using the graph-specific Louvain clustering algorithm [3]. The Louvain algorithm is suitable in our approach as, unlike most of the other clustering methods, it does not require a prior knowledge of the minimum number of clusters. For unspecified events that are not matching our training corpus, the NLP module detects frequent tags and keywords within local cells, in order to identify peaks at local and global scales.



```
{"header":{"time":"Fri Jan 01 00:39:11 +0000
2016","geo":{"type":"point","coordinates":[34.84863834,-95.54509333]}, "source":
"twitter", "eventType":["Party"], "type":"text","potentialEvent":"true",
"eventname":"Social events/party","country":"United States","city":"Oklahoma"},
"payload":{"tags":[],"id":"2605873770","followers":"581","title":"","text":"Party @
Stacey's   😂", "viewers":"0", "screenName":"TheLedgenBeare", "language":"en",
"displayName":"Ledgen","url":null}}
```

Figure 3: Sample Data Packet from Twitter Streams

As events can be discovered more efficiently on small-scale regions (starting from leaf nodes), a bottom-up approach for clustering close-by and similar events is developed, so that redundant events on different spatial resolutions can be aggregated, and their spatial scope can be upgraded. Visualization of EoIs with the same spatial resolution on maps does not make sense, since these events have different significance from spatio-temporal perspectives. For instance, events of someone's birthday cannot be displayed at a national level, except is this person is a celebrity, and that happening had spread throughout the country. Our hierarchical clustering technique for event aggregation works as follows. Starting from events at neighborhood/district level (i.e., which corresponds with leaf cells in our tree), the system clusters identical events at higher levels of abstraction, and incrementally increases their spatial scope. Local clusters are first compared with their siblings in the hierarchical tree, with the aim of aggregating and updating the scope of similar events. Merging two clusters (Cl_1, Cl_2) from two different cells ($C_{1,n}$, $C_{2,n}$) at a depth level n in the tree, will result in upgrading their spatial scope from zoom level k (e.g., corresponds to district level on map) to zoom level $k-1$ (e.g., corresponds to city level). An event cluster Cl_i is represented as follows:
$Cl_i = \langle id, ptGeom, eventClass, eventProperties, packetIDs, imageURLs, iconId, zoomLevelStart, zoomLevelEnd\rangle$

where 'id' is cluster identifier, 'pointGeom' is the centroid point location, 'eventClass' and 'eventProperties' depict the event class(es) and a list of top frequent meaningful words within the cluster, 'packetIDs' is the list of data packets identifiers forming that cluster, 'imageURLs' is the list top selected image URLs, 'iconId' is the icon identifier related to the event class, and 'zoomLevelStart, zoomLevelEnd' correspond to the multiple resolutions where this event is available to be displayed on map.

• *Hadath's query engine* supports efficient retrieval of in-memory and disk indexed events based on the main querying attributes, that are, the spatial, temporal dimensions, and the map levels of detail. The *visualizer* provides a new dimension to existing maps by illustrating extracted knowledge from live streams in the form of live events at different levels of abstraction. Figure-5 illustrates an example output of *Hadath* system by showing EoIs at different zoom levels including a) 'Grand Opening' at city-scale; b) 'Traffic Incident' at a locality-level and c) 'Birthday Party' at a neighborhood-level. The final output creates a unique and

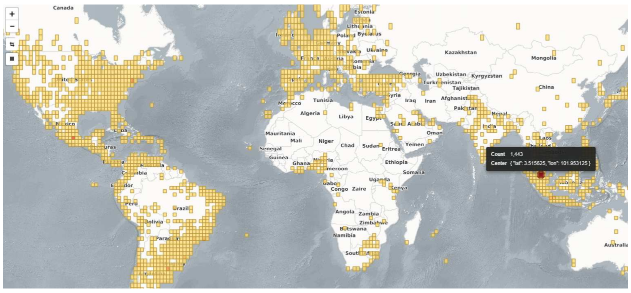


Figure 4: A snapshot of indexed data packets at a fine level of the hierarchical tree

dynamic map browsing experience.

4. DEMONSTRATION SCENARIO

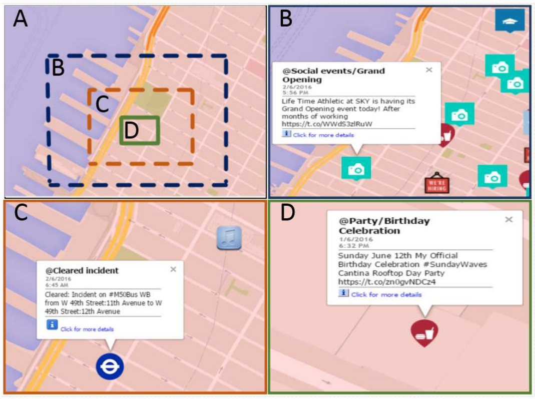
Attendees will be able to interactively use our *Hadath* system, and enjoy discovering events of different levels of abstraction on a world wide map with a smooth and fast panning and zooming capabilities. Either (near) real-time or historical events can be browsed on map with a calendar option specifying a certain time threshold. This demo is intended to show the usage and efficiency of our prototype. For this purpose, several visualizations are made possible including: i) interactive tag/word clouds of events that are dynamically adapted when changing the specified spatio-temporal scope (i.e., by zooming, panning or applying a rectangular range selection); ii) statistical plots and histograms that illustrates the number of raw data packets as well as clusters of events at different zoom levels; and finally 3) the multi-resolution event-enriched map visualization, where events of higher significance are displayed at higher abstraction levels.

5. CONCLUSION

This paper introduces a system, called *Hadath*, that builds multi-resolution event-enriched maps by handling social data streams, and by developing different algorithms for the efficient extraction, clustering, and mapping of live events. Hadath wraps incoming unstructured data streams into data packets, that is, a generic structured format of a potential event. These packets are then processed to extract EoIs based on a hierarchical clustering technique, which defines the spatio-temporal scope for each event. The system can provide valuable knowledge from crowd-sourced data to authorities, market firms, event organizers, and end-users to help in decision making. In future, we plan to merge more data sources (e.g., Flickr, online newspapers) to increase correctness and conciseness of detected events. Furthermore, an extensive performance evaluation of the different solutions need to be conducted with respect to closely-related systems.

6. REFERENCES

[1] F. Alvanaki, S. Michel, K. Ramamritham, and G. Weikum. See what's enblogue: real-time emergent topic identification in social media. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 336–347. ACM, 2012.



@Social events/Grand Opening
2/6/2016
5:56 PM
Life Time Athletic at SKY is having its Grand Opening event today! After months of working https://t.co/WWdS3zlRuW
Click for more details

@Cleared incident
2/6/2016
6:45 AM
Cleared: Incident on #M50Bus WB from W 49th Street:11th Avenue to W 49th Street:12th Avenue
Click for more details

@Party/Birthday Celebration
1/6/2016
6:32 PM
Sunday June 12th My Official Birthday Celebration #SundayWaves Cantina Rooftop Day Party https://t.co/zn0gvNDCz4
Click for more details

Figure 5: An example output of *Hadath* system. A) Overview map of the area. B) EoIs at a city-scale that are of general interest to the residents. C) EoIs at a locality-level and D) EoIs at a neighborhood-level that are progressively more specific in their spatial scope.

[2] F. Atefeh and W. Khreich. A survey of techniques for event detection in twitter. *Comput. Intell.*, 31(1):132–164, Feb. 2015.

[3] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[4] X. Dong, D. Mavroeidis, F. Calabrese, and P. Frossard. Multiscale event detection in social media. *Data Min. Knowl. Discov.*, 29(5):1374–1405, Sept. 2015.

[5] S. B. Kaleel and A. Abhari. Cluster-discovery of twitter messages for event detection and trending. *Journal of Computational Science*, 6:47–57, 2015.

[6] J. Krygier and D. Wood. *Making maps: a visual guide to map design for GIS.* Guilford Press, 2011.

[7] A. Magdy, L. Alarabi, S. Al-Harthi, M. Musleh, T. M. Ghanem, S. Ghani, and M. F. Mokbel. Taghreed: a system for querying, analyzing, and visualizing geotagged microblogs. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 163–172. ACM, 2014.

[8] F. U. Rehman, A. Lbath, M. A. Rahman, S. Basalamah, I. Afyouni, A. Ahmad, and S. O. Hussain. Toward dynamic path recommender system based on social network data. In *Proceedings of the 7th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, IWCTS '14, pages 64–69, New York, NY, USA, 2014. ACM.

[9] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[10] H. Samet, J. Sankaranarayanan, M. D. Lieberman, M. D. Adelfio, B. C. Fruin, J. M. Lotkowski, D. Panozzo, J. Sperling, and B. E. Teitler. Reading news with maps by exploiting spatial synonyms. *Commun. ACM*, 57(10):64–77, Sept. 2014.

Declarative Graph Querying in Practice and Theory

Tutorial Abstract

George H. L. Fletcher
Technische Universiteit Eindhoven
g.h.l.fletcher@tue.nl

Hannes Voigt
Technische Universität Dresden
hannes.voigt@tu-dresden.de

Nikolay Yakovets
Technische Universiteit Eindhoven
hush@tue.nl

ABSTRACT

With the recent resurgence of interest in graph data management, there has been a flurry of research on the design and engineering of graph query languages. On the design side, there is a large body of theoretical results that have been obtained regarding graph languages. On the engineering side, many sophisticated scalable solutions for graph query processing have been developed and put into practice. While both areas are focusing on the study of graph query languages, there has been relatively little work bridging the results on both sides. This tutorial will survey the state of the art in this landscape with a particular focus on uncovering and highlighting indicative research issues that are ripe for collaboration and cross-fertilization between the engineering and theoretical studies of graph database systems.

1. MOTIVATION

The mathematical concept of a graph is somewhat a rediscovered old friend in the database community. Predating relational database systems, the CODASYL network data model resembles essentially graph data. In the 1980s and early 1990s, with the rise of object-oriented programming and advent of object-oriented database systems, research considered graph-based data models and graph query languages [2]. With the continued dominance of relational DBMSs, none of these efforts got any sustainable traction in industry. In the last decade, however, the graph concept has a considerable revival with three major trends driving it.

The first driver is the Semantic Web movement [8]. The idea of the semantic web gave rise to the RDF [38] data model, which structures data as a labeled graph. This propelled the publication and maintenance of thousands of open RDF datasets on the internet, most famously DBpedia [3]. It also sparked research in every corner of the database community – ranging from works investigating the fundamental properties of query languages for labeled graphs to the design of storage structures and query engines for RDF data.

The second driver is agility with respect to the management of data. New application domains (e.g. [16, 39]) as well as novel development methods [7] increased the demand for data models that are less rigid and schema-oriented but more ad-hoc and data-oriented. Graph data models typically excel in this regard as new nodes and edges can be added anytime, regardless of their properties. This propelled the proliferation of the Property Graph model and corresponding DBMSs, such as Neo4j[1] and Apache TinkerPop Blueprints[2] implementations. By now also major DBMS vendors such as IBM and Oracle have put their weight behind the Property Graph model and are developing Property Graph-based data management solutions.

The third driver is a shift in interest of analytics from merely reporting towards data-intensive science and discovery [17]. One major method in this discipline is network analysis, which puts the focal point of interest on the connectivity of entities. The toolbox of network analysis offers a rich set of algorithms and measures. These tools give incentives to consider the graph structure of data collections in a wide range of application fields, further increasing the demand for scalable graph data management solutions.

Today, graph data management has become a major topic in the database community, in research as well as industry. There are several new challenges of graph data management which fundamentally distinguish it from tabular or nested (XML, JSON) data. An exemplification of how much traction graph data management has gained is the Linked Data Benchmark Council (LDBC).[3] In LDBC, research and industry are jointly developing standardized benchmarks for graph data management workloads to accelerate the maturing of graph management systems by increasing competition.

A rather new LDBC initiative is the Graph Query Language Standardization Task Force. The query language is one of the most crucial elements of a DBMS. It defines the functionality of a DBMS and how it is exposed to the user. At the same time, it sets the tone for the DBMS implementation by requiring certain functionality. Establishing a standardized graph query language, such as SQL for relational systems, is the next step towards more competition and progress. The task force brings together a group of researchers from engineering and theory as well as developers and representatives from industry.

One early lesson learned in the task force is that there exist two disparate bodies of work surrounding graph query lan-

[1] http://neo4j.com/

[2] http://tinkerpop.apache.org/

[3] http://ldbcouncil.org/

Series ISSN: 2367-2005

10.5441/002/edbt.2017.79

guages. One body of work focuses on the foundational issues arising in the design of graph query languages, their functionality, semantics, and formal properties such as decidability, query complexity, and containment. The other body of work focuses primarily on the engineering of systems, considering the design of storage and indexing solutions and scalable query processing engines for graph data. Due to this divide in research, it is clear that we will not get the best systems possible with the knowledge available. With both sides rather oblivious to more recent advances of the other, particularly challenges at their intersection often remain untouched.

This tutorial aims at uncovering and highlighting indicative research issues that are ripe for collaboration and cross-fertilization. In the core fields of design of declarative query languages and query processing, we will give an overview of recent advances. We will also point out their rich connections and new research challenges which arise from bringing theory and engineering together. Overall, we aim to motivate and stimulate such bridges, towards a broader coherent understanding and further improvements in the design and engineering of graph database systems.

2. SCOPE OF THE TUTORIAL

Audience. The tutorial is relevant for EDBT as well as ICDT attendees. The intended audience of the tutorial includes:

- Researchers interested in novel open challenges in graph data management or who are particularly interested in collaboration and cross-fertilization between theory and practice and want to have a kick start.
- Professionals that work in graph database system engineering and graph query language design and are interested in foundational background and broadening their scope of interesting query features.

Apart from basic knowledge about graph and database concepts there are no special requirements for this tutorial.

Coverage. The attendees of this tutorial will take home:

(i) an overview of the landscape of declarative graph query languages covering the most important features with their different functionality and properties from a practical as well as theoretical standpoint;

(ii) a survey of the foundations and recent advances accomplished by engineering and theory in graph query processing and optimization; and,

(iii) insights into open challenges for both foundational and engineering work and in particular for research topics at the intersection of both.

Scope. We scope the tutorial to core topics in graph query language design and processing. We look at the selected topics from a data management perspective, i.e., the focus is on concepts and techniques relevant to the engineering of graph data management system with a declarative query language interface. With this specific focus, there is already a wealth of results and open research challenges.

In particular, we will not discuss the design of streaming, distributed, federated, or parallel processing solutions. We will also not cover analytical topics such as graph search, graph clustering, graph pattern mining, etc. It would be impractical to also cover these topics in a focused 3-hour tutorial. Furthermore, there have been excellent tutorials on these topics recently (e.g., [19, 20, 21, 42]).

3. TUTORIAL OUTLINE

After a short illustrative introduction to the distinctive properties of graph data and graph queries, the tutorial will cover two core research areas in graph query languages: language design and query processing including data representation and query optimization aspects. Within each area, we will present the current state of the art in both theory and engineering and discuss important bridges between the bodies of work in these areas.

3.1 Graph query languages

Advances in theory. As indicated already, there is a long history of the study of graph query languages. The theoretical study of graph query languages (expressive power, evaluation complexity) has advanced ahead of the engineering of graph databases, e.g., the study of regular path queries since the 1980s. We will give a systematic presentation of the current design space of graph query languages in the theory community, including a historical perspective on this development. Major languages here include subgraph matching queries, path algebras, regular path queries, and reachability [4, 6, 10, 12, 13, 24, 34, 35, 40, 43].

Advances in engineering. With the recent proliferation of graph database system such as Neo4j, Virtuoso[4], and many others in industry and open source communities, there is a zoo of graph query languages available today. All of them offer some flavor of subgraph matching and reachability querying functionalities. We will give a structured overview of the major players in the field such as SPARQL 1.1, openCypher[5], declarative pattern matching in Gremlin, and PGQL [37] and point out their main functional and distinctive features. A look at the LDBC benchmark [11] queries will complement this to a summary of the functional features available and required from a practical, use case-driven standpoint.

Challenges. By contrasting the theoretical design space with practical query languages and use cases, we point out certain matches and mismatches, that give opportunities for knowledge transfer or give rise to new research challenges. Recently, for instance, practical query languages such as SPARQL 1.1 and openCypher have introduced support for regular path queries, which are very well studied in theory, while there is much room for aligning practical languages with this literature. Another area where many open research challenges remain is in aligning recent engineering efforts centered around the Property Graph model, on the one hand, with theoretical results applying mainly to labeled graphs, on the other. In particular, the impact which operations on graph properties might have on fundamental language properties must be considered. Finally, practical querying languages demand for functionalities is not considered in-depth from a theory perspective yet, such as aggregation queries, top-k queries, or diversity in path queries. Other aspects we will cover are: closedness/composability versus views; and, path logics versus traversal DSLs.

[4] http://virtuoso.openlinksw.com/

[5] http://www.opencypher.org/

3.2 Graph query processing

Advances in theory. The complexity of static query analysis (query containment, equivalence) is well understood for variations of graph path queries [9, 22, 34]. We will provide tutorial participants with an overview of the fundamental results for the languages surveyed in Section 3.1, with a particular focus on the demarcation between decidable and undecidable extensions of regular path queries. Tractable language-independent characterizations of graph query languages have been established in terms of the structure of a given graph instance. These characterizations are the basis for index data structures for path query evaluation/acceleration. We will provide an overview of recent advances in the theory of structural indexing and compression methods for graph data and their formal connections to the graph query languages [15, 25]. We will also discuss recent advances in the theory of worst-case optimal joins algorithms, as applied to graph query processing [1, 29].

Advances in engineering. In query processing, algorithms and graph representation go hand in hand. Node and edge tables, compressed sparse row format, and triple tables are the most common techniques used for primary graph data representation. Recent works considered various refinements of these techniques to increase efficiency by compression [1, 28], partitioning [31], triple indexing [23, 28, 44], and path indexing [14, 36]. Other advances concern the updatability of the data structures used [26, 28, 44]. Within the tutorial, we will give a crisp intro into the common base techniques and provide an overview of main ideas of the refinements. On the algorithm side, we will concentrate on advances in join processing for graph queries, since these advances are relevant for many of the query classes from the design space. In the tutorial, we will cover automata [41] and two-way join-based approaches [23, 26, 28, 33] as well as approaches that improve the utilization of high combined selectivities in graph queries, such as sideways information passing [27] and worst-case optimal n-way joins [1, 30]. We also highlight current challenges in scalability and efficiency [5].

Challenges. Research on worst-case optimal joins actually stretches from theory to engineering and excellently exemplifies the benefits of bridging both realms. We will use this example to illustrate to the tutorial participants how bridging effort can result in coherent understanding and advanced solutions. With this motivation in place, we point out further bridging challenges. For instance, while structural indexing is a very promising method from theory it has not been echoed much in system implementation. In engineering, though, updatability is an important concern, which theory is challenged to give more consideration. Further bridging challenges we will point out are n-way joins with multiset semantics and algorithmic applications of static query analysis.

3.3 Looking ahead

We will round out the tutorial with a discussion of promising research advances which have not yet bridged the gap between the theory and engineering of graph query languages. These include topics such as the decidability, complexity, and containment of graph query languages involving node and edge creation [18, 32], features which are particularly appropriate in the context of the Property Graph data model. These developments have good potential for research impact in the intersection of engineering and theoretical investigations of graph query languages.

The tutorial will conclude with a recap of the major areas that we see for collaboration and cross-fertilization between engineering and theory.

4. BIOGRAPHY OF THE PRESENTERS

George Fletcher is an associate professor of computer science at Eindhoven University of Technology. He obtained his PhD from Indiana University Bloomington in 2007. His research interests span query language design and engineering, foundations of databases, and data integration. His current focus is on management of massive graphs such as social networks and linked open data. He was a co-organizer of the EDBT Summer School on Graph Data Management (2015) and is currently a member of the LDBC Graph Query Language Standardization Task Force.

Hannes Voigt is a post-doctoral researcher at the Dresden Database Systems Group, Technische Universität Dresden and obtained his PhD from the same university in 2014. He worked on various database topics such as physical design, management of schema-flexible data, and self-adapting indexes. From 2010 to 2011, he worked at SAP Labs, Palo Alto contributing to a predecessor of SAP HANA Graph Project. His current research focuses on database evolution and versioning, declarative graph query languages and efficient graph processing on NUMA in-memory storage systems. He is also member of the LDBC Graph Query Language Standardization Task Force.

Nikolay Yakovets is an assistant professor of computer science at Eindhoven University of Technology. He obtained his PhD from Lassonde School of Engineering at York University in 2016. He worked on various database topics at IBM CAS Canada and Empress Software Canada. His current focus is on design and implementation of core database technologies, management of massive graph data, and efficient processing of queries on graphs.

5. REFERENCES

[1] C. R. Aberger, A. Nötzli, K. Olukotun, and C. Ré. EmptyHeaded: Boolean Algebra Based Graph Processing. *CoRR*, abs/1503.02368, Mar. 2015.

[2] R. Angles and C. Gutiérrez. Survey of Graph Database Models. *ACM Computing Surveys*, 40(1), 2008.

[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. DBpedia: A Nucleus for a Web of Open Data. In *ISWC/ASWC*, pages 722–735, Busan, Korea, 2007.

[4] P. B. Baeza. Querying graph databases. In *PODS*, pages 175–188, New York, NY, 2013.

[5] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. gMark: Schema-driven generation of graphs and queries. *IEEE Trans. Knowl. Data Eng.*, in press, 2017.

[6] P. Barceló, G. Fontaine, and A. W. Lin. Expressive path queries on graphs with data. In *LPAR*, pages 71–85, Stellenbosch, South Africa, 2013.

[7] K. Beck et al. Manifesto for Agile Software Development. http://agilemanifesto.org/, 2001.

[8] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Sci. Amer.*, pages 34–43, May 2001.

[9] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Reasoning on regular path queries. *SIGMOD Record*, 32(4):83–92, 2003.

[10] M. P. Consens and A. O. Mendelzon. Graphlog: a visual formalism for real life recursion. In *PODS*, pages 404–416, Nashville, 1990.

[11] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat-Pérez, M. Pham, and P. A. Boncz. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD*, pages 619–630, 2015.

[12] D. Figueira and L. Libkin. Path logics for querying graphs: Combining expressiveness and efficiency. In *LICS*, pages 329–340, Kyoto, 2015.

[13] G. H. L. Fletcher, M. Gyssens, D. Leinders, D. Surinx, J. Van den Bussche, D. Van Gucht, S. Vansummeren, and Y. Wu. Relative expressive power of navigational querying on graphs. *Inf. Sci.*, 298:390–406, 2015.

[14] G. H. L. Fletcher, J. Peters, and A. Poulovassilis. Efficient regular path query evaluation using path indexes. In *EDBT*, pages 636–639, Bordeaux, 2016.

[15] G. H. L. Fletcher et al. Similarity and bisimilarity notions appropriate for characterizing indistinguishability in fragments of the calculus of relations. *J. Log. Comput.*, 25(3):549–580, 2015.

[16] M. J. Franklin, A. Y. Halevy, and D. Maier. From Databases to Dataspaces: A New Abstraction for Information Management. *SIGMOD Record*, 34(4):27–33, 2005.

[17] T. Hey, S. Tansley, and K. M. Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery.* Microsoft Research, 2009.

[18] J. Hidders. *A graph-based update language for object-oriented data models.* PhD thesis, Eindhoven University of Technology, 2001.

[19] Y. Ke, J. Cheng, and J. X. Yu. Querying large graph databases. In *DASFAA*, pages 487–488, 2010.

[20] A. Khan and S. Elnikety. Systems for big-graphs. *PVLDB*, 7(13):1709–1710, 2014.

[21] A. Khan, Y. Wu, and X. Yan. Emerging graph queries in linked data. In *ICDE*, pages 1218–1221, 2012.

[22] E. V. Kostylev, J. L. Reutter, and D. Vrgoc. Containment of data graph queries. In *ICDT*, pages 131–142, Athens, Greece, 2014.

[23] Y. Luo, F. Picalausa, G. H. L. Fletcher, J. Hidders, and S. Vansummeren. Storing and indexing massive RDF datasets. In R. De Virgilio, F. Guerra, and Y. Velegrakis, editors, *Semantic Search over the Web*, pages 31–60. Springer Berlin Heidelberg, 2012.

[24] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Strong simulation: Capturing topology in graph pattern matching. *ACM Trans. Database Syst.*, 39(1):4, 2014.

[25] S. Maneth and F. Peternek. A survey on methods and systems for graph compression. *CoRR*, abs/1504.00616, 2015.

[26] B. Motik, Y. Nenov, R. Piro, I. Horrocks, and D. Olteanu. Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In *AAAI*, pages 129–137, 2014.

[27] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, pages 627–640. ACM, 2009.

[28] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

[29] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2013.

[30] D. T. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra. Join Processing for Graph Patterns: An Old Dog with New Tricks. In *GRADES*, pages 2:1–2:8, 2015.

[31] M. Paradies, W. Lehner, and C. Bornhövd. GRAPHITE: An Extensible Graph Traversal Framework for Relational Database Management Systems. In *SSDBM*, pages 29:1–29:12. ACM, 2015.

[32] J. Paredaens, P. Peelman, and L. Tanca. G-Log: A graph-based query language. *IEEE Trans. Knowl. Data Eng.*, 7(3):436–453, 1995.

[33] R. Raman, O. van Rest, S. Hong, Z. Wu, H. Chafi, and J. Banerjee. PGX.ISO: Parallel and Efficient In-Memory Engine for Subgraph Isomorphism. In *GRADES*, pages 1–6. ACM, 2014.

[34] J. L. Reutter, M. Romero, and M. Y. Vardi. Regular queries on graph databases. In *ICDT*, pages 177–194, Brussels, 2015.

[35] S. Santini. Regular languages with variables on graphs. *Inf. Comput.*, 211:1–28, 2012.

[36] J. Sumrall, G. H. L. Fletcher, A. Poulovassilis, J. Svensson, M. Vejlstrup, C. Vest, and J. Webber. Investigations on path indexing for graph databases. In *PELGA*, Grenoble, 2016.

[37] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. PGQL: a property graph query language. In *GRADES*, Redwood Shores, CA, 2016.

[38] W3C. RDF 1.1 Concepts and Abstract Syntax. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/, Feb. 2014.

[39] H. Werner, C. Bornhövd, R. Kubis, and H. Voigt. MOAW: An Agile Visual Modeling and Exploration Tool for Irregularly Structured Data. In *BTW*, pages 742–745, 2011.

[40] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.

[41] N. Yakovets, P. Godfrey, and J. Gryz. Query planning for evaluating SPARQL property paths. In *SIGMOD*, pages 1875–1889, San Francisco, 2016.

[42] D. Yan, Y. Bu, Y. Tian, A. Deshpande, and J. Cheng. Big Graph Analytics Systems. In *SIGMOD*, pages 2241–2243, 2016.

[43] J. Yu and J. Cheng. Graph reachability queries: A survey. In C. C. Aggarwal and H. Wang, editors, *Managing and Mining Graph Data*, Advances in Database Systems, pages 181–215. Springer US, 2010.

[44] L. Zou, M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao. gstore: a graph-based SPARQL query engine. *VLDB J.*, 23(4):565–590, 2014.

Multi-model Data Management: What's New and What's Next?

Jiaheng Lu
Department of Computer Science
University of Helsinki, Finland
jiaheng.lu@helsinki.fi

Irena Holubová*
Department of Software Engineering
Charles University, Czech Republic
holubova@ksi.mff.cuni.cz

ABSTRACT

As more businesses realized that data, in all forms and sizes, is critical to making the best possible decisions, we see the continued growth of systems that support massive volume of non-relational or unstructured forms of data. Nothing shows the picture more starkly than the Gartner Magic quadrant for operational database management systems, which assumes that, by 2017, all leading operational DBMSs will offer multiple data models, relational and NoSQL, in a single DBMS platform. Having a single data platform for managing both well-structured data and NoSQL data is beneficial to users; this approach reduces significantly integration, migration, development, maintenance, and operational issues. Therefore, a challenging research work is how to develop efficient consolidated single data management platform covering both relational data and NoSQL to reduce integration issues, simplify operations, and eliminate migration issues. In this tutorial, we review the previous work on multi-model data management and provide the insights on the research challenges and directions for future work. The slides and more materials of this tutorial can be found at http://udbms.cs.helsinki.fi/?tutorials/edbt2017.

1. INTRODUCTION

In recent years the term big data has become a phenomenon that breaks down borders of many technologies and approaches that have so far been acknowledged as mature and robust for any conceivable application. One of the most challenging issues is the "*Variety*" of the data. It may be presented in various types and formats – structured, semi-structured and unstructured – and produced by different sources, and hence natively have various models.

To address the Variety challenge, probably the first type of respective specific database management systems (DBMS) are *NoSQL databases* [34] which can be further classified[1] to *soft* (e.g., object or XML DBMSs), and *core* (e.g., key/value, document, column, or graph DBMSs). From another point of view we can classify them to *single-model* and *multi-model*. The latter type enables to store and process structurally different data, i.e. data with distinct models, which corresponds to the Variety aspect of big data. This approach can be considered as an opposite idea to the "*One Size Does Not Fit All*" argument [39]. However, it can be also understood as a way of re-architecting traditional database models, namely the relational model, to handle new database requirements that were not present during its establishment decades ago [24]. Nothing shows the picture more starkly than the Gartner Magic quadrant for operational database management systems [18], which assumes that, by 2017, all leading operational DBMSs will offer multiple data models, relational and NoSQL, in a single DBMS platform.

In this tutorial, we review the previous work on multi-model data management and give insights on the research challenges and opportunities. First, we show that the idea of multi-model DBMSs is not a brand new approach. It can be traced back to Object-Relational Data Management Systems (ORDBMS) in the early 1990s and in a more broader scope even to federated and integrated DBMSs in the early 1980s. An ORDBMS system can manage different types of data such as relational, object, text and spatial by plugging domain specific data types, functions and index implementations into the DBMS kernels. For instance, PostgreSQL [6] can store relational, spatial and XML data. Recently, we can observe a new trend among NoSQL databases in the support of multiple data models against a single, integrated backend, while meeting the growing requirements for scalability and performance. For example, OrientDB [7] is a graph database extended to support multi-model queries, while ArangoDB [10] is moving from purely document model to the support of also key-value, graph and JSON data.

Second, we dive in three key aspects of technology in a multi-model database system including (1) storage strategies for multi-model data; (2) query languages accessing data across multiple models; and (3) query evaluation and its optimization in the context of multiple data models.

Finally, we provide comparison of features of the existing multi-model DBMSs and we discuss related open problems and remaining challenges.

To the best of our knowledge this is the first tutorial to discuss the state-of-the-art research works and industrial trends in the context of multi-model data management. Recent tutorials related to the big data world include SQL-on-Hadoop Systems [12], open-source on big data [16], knowledge bases

*Supported by the MŠMT ČR grant PROGRES.

[1] http://nosql-database.org/

Series ISSN: 2367-2005

10.5441/002/edbt.2017.80

in big data analytics [40], or big time-series data management [35], i.e., different aspects of big data challenges.

2. COVERED TOPICS

2.1 Background, History and Classification

In the first part of the tutorial we first provide a motivating example of a multi-model application and briefly describe most common data models used in the world of multi-model DBMSs (mainly key/value, relational, JSON, XML, and graph). Next, we focus on their history and classification.

The world of multi-model DBMSs can be divided into *single-database* and *multi-database* (see Figure 1), depending on whether the multiple models are handled in a single DBMS or there exist a number of cooperating or centrally managed DBMSs, each handling own data model(s).

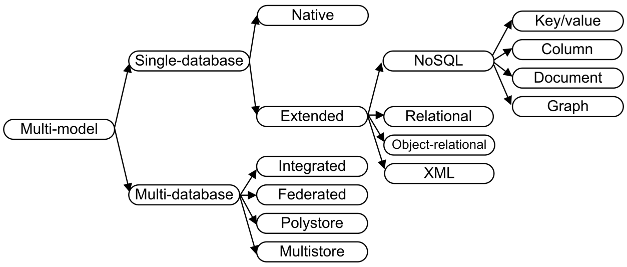


Figure 1: Classification of multi-model data management systems

The first approaches towards multi-model multi-database data management can be seen in *integrated DBMSs* [37] and *federated DBMSs* [20, 36]. Both types of systems can be characterized as a meta-DBMS consisting of a collection of (possibly) heterogeneous DBMSs which can differ in data models, constraints, query languages, and/or transaction management. The data integration is usually based on the idea of *mediators* [43]. The main difference is that in federated systems the DBMSs are autonomous and cooperate. Thus federated databases provide a compromise between no integration (where the users must explicitly interface with multiple autonomous DBMSs) and total integration (where the users can access data through a single global interface but cannot directly access a DBMS as a local user) [36].

Recently there has appeared a successor of federated databases – so-called *polystore systems* [38]. The key representative, system BigDAWG [17], also enables users to pose declarative queries that span several DBMSs. However, it consists of *islands of information*, i.e. collections of DBMSs accessed with a single query language (e.g., relational or array). Cross-island queries are supported using casting (e.g., tables to arrays or vice versa).

Another recent related approach from the area of big data analytics represent so-called *multistore systems* [23, 44]. For example system MISO [23] involves two types of data stores – a parallel relational data warehouse and a system for massive data storage and analysis (namely HDFS with Apache Hive). The aim is to combine their capabilities in order to gain more efficient query processing.

Multi-model single-database DBMSs can also be further classified. Probably the most natural classification is according their origin [2] (see Figure 1). Similarly to XML databases, we can distinguish *native* and *extended* DBMSs depending on whether the support for multiple models was the initial feature of the system, or it was added later. In the latter case we can find representatives amongst all four core types of NoSQL databases as well as traditional DBMS.

2.2 Overview and Comparison

In the second part of the tutorial we take a closer look at particular multi-model single-database DBMSs from the point of view of three key aspects of a database system.

The first database challenge is to develop a strategy to store distinct data models. Approaches used in the existing multi-model DBMSs can be classified according to the combination of used models. The main group (systems such as, e.g., PostgreSQL or Microsoft SQL Server [9]) is naturally represented by the (object-)relational model extended towards other data models, such as JSON, XML etc. From the set of NoSQL databases we can observe the tendency towards multi-model data management among column stores [4], key/value stores [11], or graph databases [7]. And there are also representatives of native hierarchical data stores [5] which support other types of data models.

The second database challenge is a query language capable of accessing and combining data having distinct models. Naturally, having a single language for managing queries over both (semi-)structured and NoSQL data is convenient to users. And again, in general, this is not a new feature of a query language, as we can see, e.g., in the case of the SQL/XML [21] extension of SQL. Most of the current NoSQL multi-model databases across the spectrum of storage strategies [6, 4, 7] support an SQL-like language. However, as we will show, despite this approach is natural and user-friendly, there are significant differences as well as persisting limitations. There also exist XML or JSON query language extensions towards other data models (e.g., MarkLogic's XPath for JSON [3]), as well as specific languages like, e.g., SQL++[31], JSONiq [33], or FSD domain-specific language [24]. In a more broader scope paper [32] identifies a subset of SQL for access to NoSQL systems or paper [13] evaluates the possibilities of using declarative structures in NoSQL data processing. We also discuss other techniques, like, e.g., [14, 32, 41].

The third challenge corresponds to query evaluation and optimization. As expected, the world of multi-model DBMSs exploits and extends verified database approaches such as indices (B+ tree, inverted, range, spatial, full text, etc.), views and materialization, hashing etc. In this part of the tutorial we overview and compare the query optimization technologies used in the previously discussed systems. We also introduce the related area of benchmarking multi-model database systems. As more and more platforms are proposed to deal with multi-model data, it becomes important to have benchmarks specific for this next generation of database systems. We mention several systems for benchmarking big data systems including YCSB [15], TPCx-BB [19], Bigframe [22], and UniBench [25].

We conclude this part with comparison of features of the state-of-the-art systems in the form of system-feature matrices and a timeline demonstrating their evolution.

2.3 Open Problems and Challenges

In the last part of the tutorial we focus on open problems that must be addressed to ensure the success of multi-model DBMSs. The key areas to be discussed involve:

- Unified query processing and index structures,
- Multi-model main memory structure,
- Multi-model schema extraction, design, and optimization, especially in the context of schema-less DBMSs,
- Evolution management and model extensibility,
- Benchmarking and standardization.

In each of these areas we first briefly overview the solutions in the world of single-model DBMSs as well as eventually existing (partial) solutions among multi-model DBMSs. Then we explain the related problems in the context of multi-model databases, eventually with existing preliminary solutions. We assume that this part will raise questions to be discussed in the end of the tutorial.

3. TUTORIAL ORGANIZATION

The tutorial is planned for 1.5 hours and will have the following structure:

Motivation (5'). We motivate the need for multi-model data management by several examples in the era of big data.

History and classification (10'). We introduce the history and classification of multi-model databases, including ORDBMS [9], NoSQL databases [7, 10] and Polyglot persistence [38, 43].

Multi-model data storage (10'). We introduce various methods to store multi-model data, including object-relational model, graph model, document model and native hierarchical model.

Multi-model data query languages (15'). We compare languages for multi-model data processing, such as AQL [10], SQL++ [31], OrientDB SQL [7], and SQL/XML [21].

Multi-model query processing (15'). We overview the multi-model extensions of traditional query processing appraoches and indexes, such as B+ tree [1, 30], inverted index [8], schema discovery [42, 24], and cross-model query processing [10, 7].

Multi-model database benchmarking (15'). We introduce the previous and on-going benchmark systems for multi-model data, such as TPCx-BB [19], Bigframe [22], YCSB [15], or UniBench [25].

Open problem and challenges (20'). We conclude with a discussion of open problems and challenges for database research in the area of multi-model data management [29].

4. GOALS OF THE TUTORIAL

4.1 Learning Outcomes

The main learning outcomes of this tutorial are as follows:

- Motivation, classification and historical evolution of multi-model DBMSs.
- An overview of technologies and algorithms used by the current multi-model DBMSs including storing, query languages, and query optimization.
- Comparison of features of current multi-model DBMSs.
- A discussion of research challenges and open problems of multi-model data management.

4.2 Intended Audience

This tutorial is intended for a wide scope of audience, e.g. for developers and architects to get insights from the emerging industrial trends and its connections to scientific research, for stakeholders to make wise and informed decisions on investments in multi-model DBMS products, for motivated researchers and developers to select new topics and contribute their expertise on multi-model data, and, of course, for new developers and students to quickly gain a comprehensive picture and understand the new trends and the state-of-art techniques in this field.

Basic knowledge in relational and NoSQL databases is sufficient to follow the tutorial. Some background in semi-structured and graph query optimization would be useful, but is not necessary.

5. SHORT BIBLIOGRAPHIES

Jiaheng Lu is an Associate Professor at the University of Helsinki, Finland. He received Ph.D. degree at the National University of Singapore in 2007. He did two-year Post-doctoral research at the University of California, Irvine. His main research interests lie in the big data management and database systems, and specifically in the challenge of efficient data processing from real-life, massive data repository and Web. He has published more than sixty journal and conference papers. He has extensive experiences of the industrial cooperations with IBM, Microsoft and Huawei for the projects of NoSQL databases and performance tuning on distributed systems. He has published several books, on XML [27], Hadoop [28] and NoSQL databases [26]. His book [28] on Hadoop is one of the top-10 best-selling books in the category of computer software in China in 2013.

Irena Holubová is an Associate Professor at the Charles University, Prague, Czech Republic, where she received Ph.D. degree in 2007. Her current main research interests include big data management and NoSQL databases, big data generators and benchmarking, evolution and change management of database applications, analysis of real-world data, and schema inference. She has published more than 80 conference and journal papers; her works gained 4 awards. She has also published 2 books on XML technologies and NoSQL databases. She serves as an independent expert for evaluation and monitoring of EU FP7 and H2020 projects.

6. REFERENCES

[1] *Improving Secondary Index Write Performance in 1.2.* DataStax, Inc., 2013.

[2] *Neither Fish Nor Fowl: the Rise of Multi-Model Databases.* The 451 Group, 2013.

[3] *Application Developer's Guide – Chapter 18 Working With JSON.* MarkLogic Corporation, 2016.

[4] *Cassandra: Manage Massive Amounts of Data, Fast, without Losing Sleep.* The Apache Software Foundation, 2016.

[5] *MarkLogic: The World's Best Database for Integrating Data From Silos.* MarkLogic Corporation, 2016.

[6] *The Official Site for PostgreSQL, the World's Most Advanced Open Source Database.* The PostgreSQL Global Development Group, 2016.

[7] *OrientDB – a 2nd Generation Distributed Graph Database.* OrientDB, 2016.

[8] *PostgreSQL 9.5.3 Documentation – Chapter 61. GIN Indexes.* The PostgreSQL Global Development Group, 2016.

[9] *SQL Server 2016.* Microsoft, 2016.

[10] *Three major NoSQL data models in one open-source database.* ArangoDB, 2016.

[11] *Vertica Advanced Analytics.* Hewlett Packard Enterprise, 2016.

[12] D. Abadi, S. Babu, F. Ozcan, and I. Pandis. Tutorial: SQL-on-Hadoop Systems. *PVLDB*, 8(12):2050–2061, 2015.

[13] M. Bach and A. Werner. Standardization of NoSQL Database Languages. In *BDAS*, pages 50–60, 2014.

[14] F. Bugiotti, L. Cabibbo, P. Atzeni, and R. Torlone. Database Design for NoSQL Systems. In *ER*, pages 223–231, 2014.

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, pages 143–154, 2010.

[16] C. Douglas and C. Curino. Blind Men and an Elephant Coalescing Open-source, Academic, and Industrial Perspectives on BigData. In *ICDE*, pages 1523–1526, 2015.

[17] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. The BigDAWG Polystore System. *SIGMOD Rec.*, 44(2):11–16, Aug. 2015.

[18] D. Feinberg, M. Adrian, N. Heudecker, A. M. Ronthal, and T. Palanca. Gartner Magic Quadrant for Operational Database Management Systems, 12 October 2015.

[19] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H. Jacobsen. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In *ACM SIGMOD*, pages 1197–1208, 2013.

[20] M. Hammer and D. McLeod. *On Database Management System Architecture.* MIT/LCS/TM. Mass. Inst. of Technology, Laboratory for Computer Science, 1979.

[21] ISO. ISO/IEC 9075-14:2011 Information technology – Database languages – SQL – Part 14: XML-Related Specifications (SQL/XML), 2011.

[22] M. Kunjir, P. Kalmegh, and S. Babu. Thoth: Towards Managing a Multi-System Cluster. *PVLDB*, 7(13):1689–1692, 2014.

[23] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey. MISO: Souping Up Big Data Query Processing with a Multistore System. In *ACM SIGMOD*, pages 1591–1602, 2014.

[24] Z. H. Liu and D. Gawlick. Management of Flexible Schema Data in RDBMSs – Opportunities and Limitations for NoSQL. In *CIDR*, 2015.

[25] J. Lu. Towards Benchmarking Multi-Model Databases http://udbms.cs.helsinki.fi/?projects/ubench. In *CIDR*, 2017.

[26] J. Lu. *Big data challenge and NoSQL databases.* House of Electrical Industry in China, ISBN:978-7-121-19660-7, 423 pages, April, 2013.

[27] J. Lu. *An Introduction to XML Query Processing and Keyword Search.* Springer Berlin Heidelberg, ISBN: 978-3-642-34554-8, 201 pages, March 16, 2013.

[28] J. Lu. *Programming on Hadoop.* China Industrial Press, ISBN: 978-7-111-35944-9, 441 pages, October, 2011.

[29] J. Lu, Z. H. Liu, P. Xu, and C. Zhang. UDBMS: road to unification for multi-model data management. *CoRR*, abs/1612.08050, 2016.

[30] P. E. O'Neil. The SB-tree: An Index-sequential Structure for High-performance Sequential Access. *Acta Inf.*, 29(3):241–265, June 1992.

[31] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ Unifying Semi-structured Query Language, and an Expressiveness Benchmark of SQL-on-Hadoop, NoSQL and NewSQL Databases, 2016.

[32] J. Rith, P. S. Lehmayr, and K. Meyer-Wegener. Speaking in Tongues: SQL Access to NoSQL Systems. In *SAC*, pages 855–857, 2014.

[33] J. Robie, G. Fourny, M. Brantner, D. Florescu, T. Westmann, and M. Zaharioudakis. The JSON Query Language, 2016.

[34] P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.* Addison-Wesley Professional, 1st edition, 2012.

[35] Y. Sakurai, Y. Matsubara, and C. Faloutsos. Mining and Forecasting of Big Time-series Data. In *Proceedings of the 2015 ACM SIGMOD*, pages 919–922, 2015.

[36] A. P. Sheth and J. A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Comput. Surv.*, 22(3):183–236, Sept. 1990.

[37] J. M. Smith, P. A. Bernstein, U. Dayal, N. Goodman, T. Landers, K. W. T. Lin, and E. Wong. Multibase: Integrating Heterogeneous Distributed Database Systems. In *AFIPS '81*, pages 487–499, New York, NY, USA, 1981. ACM.

[38] M. Stonebraker. The Case for Polystores, 2015.

[39] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE '05*, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[40] F. M. Suchanek and G. Weikum. Knowledge Bases in the Age of Big Data Analytics. *PVLDB*, 7(13):1713–1714, 2014.

[41] D. Tahara, T. Diamond, and D. J. Abadi. Sinew: a SQL System for Multi-structured Data. In *SIGMOD*, pages 815–826, 2014.

[42] D. A. Teich. Database Schemas Still Needed, Despite Hadoop and NoSQL Pretensions, 2016.

[43] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *Computer*, 25(3):38–49, Mar. 1992.

[44] Y. Xu, P. Kostamaa, and L. Gao. Integrating Hadoop and Parallel DBMs. In *SIGMOD '10*, pages 969–974, New York, NY, USA, 2010. ACM.

Data Security and Privacy for Outsourced Data in the Cloud

Cetin Sahin and Amr El Abbadi
Department of Computer Science
University of California, Santa Barbara
{cetin, amr}@cs.ucsb.edu

ABSTRACT

Although outsourcing data to cloud storage has become popular, the increasing concerns about data security and privacy in the cloud blocks broader cloud adoption. Ensuring data security and privacy, therefore, is crucial for better and broader adoption of the cloud. This tutorial provides a comprehensive analysis of the state-of-the-art in the context of data security and privacy for outsourced data. We aim to cover common security and privacy threats for outsourced data, and relevant novel schemes and techniques with their design choices regarding security, privacy, functionality, and performance. Our explicit focus is on recent schemes from both the database and the cryptography and security communities that enable query processing over encrypted data and access oblivious cloud storage systems.

1. INTRODUCTION

Recent advances in cloud technologies have made outsourcing personal and corporate data to cloud storage servers increasingly popular and attractive, due to its promise of high scalability and availability. However, this increase in utility comes with a risk of exposing data to a number of security threats. For example, a curious administrator might snoop on private data or an adversary might gain unauthorized access to sensitive information. Therefore, potential customers remain skeptical about joining the cloud due to existing confidentiality and privacy concerns [17]. For broader adoption of cloud services, concerns about data security and privacy must be addressed. The question here is how to ensure security and privacy of outsourced data while maintaining the ability to execute queries efficiently.

Providing secure and privacy-preserving data services over outsourced data is challenging. Both the database and the cryptography communities have shown great interest in providing privacy-preserving and secure data services, but there is no one scheme that solves all the security and privacy problems. Different schemes and models have different security and privacy guarantees, and these protection guarantees come at a cost: decrease in performance and functionality. There is an obvious trade-off between security/privacy and functionality/performance. Sacrificing functionality and performance completely for the sake of security and privacy makes outsourcing services impractical. Therefore, any data related service needs to seek a proper balance in the space of security, privacy, functionality and performance. In this tutorial, we aim to cover common security and privacy threats for outsourced data, and relevant state-of–the-art solutions from the database and the cryptography literature. We also discuss their limitations, open problems and further research directions for secure and private cloud storage systems. This tutorial explicitly focuses on the ability to query data in a cloud storage, while maintaining data confidentiality and access privacy.

2. TUTORIAL OUTLINE

This tutorial presents recent schemes from both the database and the cryptography and security communities in the context of outsourced data in the cloud. In particular, we focus on two aspects of outsourced data in the cloud: query processing over encrypted data and access oblivious cloud storage systems. The tutorial consists of three main sections: 1) security and privacy threats for outsourced data, 2) query processing over encrypted data, and 3) access privacy for oblivious storage. The tutorial is intended to last 3 hours. The initial section highlights security and privacy concerns for outsourced data services. The next sections provide a broad survey of research in the area concerning security/privacy models, proposed techniques/schemes, and associated problems and challenges.

2.1 Security and Privacy Threats in the Cloud

The cloud is a popular and tempting attack target. It hosts many businesses at different scales using a shared infrastructure. When an attacker attacks the cloud, it has access to consolidated data, which can have great financial value. To develop secure and privacy-preserving systems, the system designers must first develop a clear understanding of the possible threats. Therefore, the tutorial starts with a general overview of possible security and privacy threats in the context of storage services. The cloud service is assumed to be untrusted. Any unauthorized access or the cloud provider will be considered as an honest-but-curious *adversary*, where the adversary runs the protocol correctly, but may try to learn as much as possible about data. After highlighting possible security and privacy threats, to draw attention to the significance of the concerns, we will cover a

Series ISSN: 2367-2005

10.5441/002/edbt.2017.81

few recent data breaches in terms of their vulnerabilities and consequences [1, 2]. Security and privacy are required, but performance and functionality are also essential for cloud storage systems and these conflict with security and privacy requirements. The question that concludes the section is "What is the proper balance between privacy, security, functionality, and performance?".

2.2 Query Processing over Encrypted Data

Storing encrypted data in a hostile environment provides strong data confidentiality. However, the ability to perform practical query processing on encrypted data remains a major challenge. Both the database and the cryptography research communities have shown great interest in querying encrypted data including keyword search [45, 14], equality queries [51], range queries [28, 30], and order preserving encryption [5, 37]. These methods sacrifice some degree of data confidentiality for more effective querying on encrypted data and provide different levels of security guarantees. Other proposals sacrifice query efficiency for stronger data confidentiality. Examples include homomorphic encryption and predicate encryption, which enable numerical computations on encrypted data without the need for decryption [21, 22, 34]. These have been shown to be quite expensive, and thus not practical [43].

Recent tutorials that appear in VLDB, ICDE and SIGMOD [4, 3, 41, 7] present detailed surveys of systems that perform query processing on encrypted data. In this tutorial, our approach is slightly different from these earlier works. We cover concepts that have seen significant interest recently in the security and the cryptography communities such as *Symmetric Searchable Encryption* (SSE). We revisit some important privacy and security concepts and cover important papers from the main security venues like S&P and CCS while still presenting recent results in the database community.

Initially, various primitive encryption schemes are introduced since they form the building blocks for other system developments. The functionality and security guarantees of non-deterministic and deterministic encryption scheme are presented using *Advanced Encryption Standard* (AES) [38]. Homomorphic encryption provides a desirable and interesting feature which allows computations directly over encrypted data. However, to date, only specific functionality, e.g. aggregation, can be performed efficiently. The need for different encryption schemes for specific tasks has resulted in various proposals such as order preserving encryption [5] and encrypted keyword search [45]. Both the database and the cryptography communities still show great interest in developing more efficient schemes for specific tasks.

Keyword search over encrypted data has received considerable attention in the cryptography and the security communities as well as the database community. Song et al. [45] propose a foundational technique for keyword search, also known as the first SSE scheme. This work has been followed upon by various competing new security definitions and constructions in the context of SSE [23, 19, 16, 32, 12, 39]. In this part of the tutorial, we start with [19] which provides security definitions for SSE for both adaptive and non-adaptive adversarial settings and proposes constructions for both adversarial settings. In recent work, Cash et al. [12] introduce a dynamic SSE solution which supports the modification of data. It supports storing large data and has optimal and parallelizable search complexity. Another dynamic SSE solution is proposed by Naveed et al. [39] and is based on a notion of *Blind Storage*. In an interesting study, Cash et al. also show that it is possible to extend the SSE approach to handle boolean queries in [13]. We discuss how such an extension might be a guide for further developments in different contexts.

Range queries are widely used as fundamental database operations to retrieve records between an upper and a lower boundary (e.g., retrieving students who have grades between A and B). A canonical SQL query for such a query is "select * from students where grade $\leq$ B and grade $\geq$ A". In spite of its wide utilization, performing range queries in a privacy-preserving manner is still challenging. Agrawal et al. introduce *order preserving encryption* (OPE) [5] to support range queries efficiently. Unfortunately, OPE is vulnerable to statistical attacks and is limited in terms of further modifications. Since it was first proposed, there have been a large number of proposals that aim to provide more secure solutions while still being efficient [29, 37, 10, 35, 33, 20]. Modular order preserving encryption (MOPE) [10] adds a secret offset to the data before encryption to shift the ciphertext (in a ring), and to hide the real location of the encrypted data in their distribution. In [37], an improved version of MOPE has been proposed. It uses fake queries over the gap between the maximum and minimum values to improve the security of MOPE against attacks that analyze the query patterns to detect the max/min values among the encrypted data. Improvements in SSE have also benefited the database community. Similar to [13], which handles boolean queries by extending SSE, Demertzis et al. [20] recently proposed a range query solution that uses SSE. To take advantage of SSE, Demertzis et al. propose three types of indexing approaches with different space requirements in terms of domain size: quadratic, linear and logarithmic. We again discuss the proposed schemes in terms of their computational and space overheads, supported functionality, and security guarantees.

We finish this section of the tutorial by discussing full-fledged secure systems [8, 40, 6, 49]. CryptDB [40] is a secure system that processes different types of database queries using layers of different encryption mechanisms and removes layers of encryption to an appropriate layer for solving a specific query. MONOMI [49] follows CryptDB's approach of using different encryption schemes for specific queries. On the other hand, it is designed for executing analytical queries. Cipherbase [6] and TrustedDB [8] are full-fledged database system proposals that benefit from secure hardware. We discuss the advantages and disadvantages along with the security guarantees of these systems.

2.3 Oblivious Storage

Although it is necessary, encryption alone is not sufficient to solve all privacy challenges posed by the outsourcing of private data. Indeed, if *access patterns* are *not* hidden from the cloud provider, the provider could detect, for example, whether and when the same data item is repeatedly accessed, even if it does not learn the actual content of the item. This is a real threat to the privacy of outsourced data, as data access patterns can leak sensitive information using prior knowledge. For example, Islam et al. [31] showed a concrete inference attack against an encrypted e-mail repository exploiting access patterns alone. *Oblivious RAM* (ORAM) –

a cryptographic primitive originally proposed by Goldreich and Ostrovsky [24, 25] as a solution for software protection – is the standard approach to make access patterns *oblivious*. ORAM shuffles and re-encrypts data in each data access, making access patterns from any two equally long sequences of read/write operations completely indistinguishable. Hiding access patterns was initially considered in the context of memory access [25]. While classical ORAM schemes with small client memory apply directly to the memory access setting, in cloud applications a client has more storage space and is capable of storing more data locally and more importantly can outsource the storage of a large dataset to the cloud. The novel features and fast adoption of the cloud gave impetus to the research community to develop new secure data services in the past several years and many ORAM schemes have been constructed for secure cloud storage systems [11, 36, 50, 47, 46, 9, 42]. Recent works from both the database and cryptography literature present a comprehensive analysis of ORAM schemes as oblivious cloud storage [9, 42, 15].

This section of the tutorial starts with the definition of access patterns. We explicitly define the notion of securing an access pattern. This is followed by a famous attack by Islam et al. [31] that shows how the leakage of access patterns can be harmful to sensitive data. Why should we care about access patterns? Why do we need to achieve *oblivious access*? After the motivation, we move to the details of ORAM constructions, which ensure oblivious accesses. To date, two main types of ORAM constructions exist: *hierarchical* and *tree-based*. The first hierarchical ORAM to be discuss is GO-ORAM [25]. Follow-up hierarchical ORAM constructions improve different aspects of GO-ORAM such as reduced overhead and faster shuffling [27, 26]. Next, we cover the tree-based ORAM constructions which have been proposed relatively recently and extended in a large number of works [44, 48, 18]. Tree-based constructions organize the memory as a tree. The current state-of-the-art construction, Path ORAM [48], will be covered as a prototype of tree-based ORAMs. Both GO-ORAM and Path ORAM were designed for a single client and such systems do not fit the requirements of cloud deployments, since accesses to the storage are performed *sequentially*. Therefore, after explaining the building blocks of single client hierarchical and tree-based ORAMs, we will discuss how to construct ORAMs in such a way that they simulate real-world storage scenarios by inheriting features like multi-client concurrent access, asynchronicity, and, of course, security.

PrivateFS by Williams et al. [50] increases the throughput of storage by enabling parallel accesses to the storage. We present the PrivateFS framework and then focus on how it allows multiple clients to obliviously access data in parallel along with its limitations. Follow-up improvements for more practical oblivious storage schemes [46, 9, 42] will be considered in the context of system design, performance, correctness and security. Stefanov and Shi propose *ObliviStore* [46] which provides a definition for asynchronous ORAM and introduces a proxy based approach where the proxy mediates the communication between clients and the server. In a recent study, Bindschaedler et al. [9] present a subtle security issue in ObliviStore and propose a modular oblivious storage system, called *CURIOUS*. In our recent work [42], we show that the security definition used by both ObliviStore and CURIOUS does not capture asynchrony when multiple clients access storage concurrently in a realistic deployment scenario. We, therefore, propose TaoStore, a new *tree-based* ORAM scheme that processes client requests concurrently and asynchronously in a non-blocking fashion.

At the end of this section, we provide a detailed analysis of the current state of secure cloud storage, the open problems and challenges, and further research directions towards providing more practical oblivious cloud storage systems.

3. INTENDED AUDIENCE

This tutorial aims to provide a broad survey on data security and privacy, and is intended to be beneficial for anyone interested in data security and privacy. We intend to introduce to the database community state-of-the-art results from the security literature that are particularly relevant for databases. The tutorial is self-contained and does not require any prior knowledge about data security and privacy.

4. BIOGRAPHY

Amr El Abbadi is a Professor of Computer Science at the University of California, Santa Barbara. His research interests lie in the broad area of scalable database and distributed systems. El Abbadi is a Fellow of the ACM, IEEE, and AAAS and was chair of the Computer Science Department at UCSB from 2007 to 2011. He has held visiting professor positions at the University of Campinas in Brazil, IBM Almaden Research Center, the Swedish Institute of Computer Science in Stockholm, and at the University of Rennes in France.

Cetin Sahin earned his master's degree in Computer Science from UCSB in 2016. He is currently a PhD candidate at the same institute. His research interests include data security and privacy. He was a summer research assistant at NEC Laboratories in 2013 and 2014.

ACKNOWLEDGEMENTS

This work is partly funded by NSF grants CNS-1528178 and CCF-1442966.

5. REFERENCES

[1] Dropbox breach from 2012 comes back to haunt users. https://www.identityforce.com/blog/2016-data-breaches.

[2] Yahoo data breach: Almost 500 million affected. https://www.identityforce.com/blog/yahoo-data-breach-almost-500-million-affected.

[3] D. Agrawal, A. E. Abbadi, and S. Wang. Secure and privacy-preserving database services in the cloud. In *ICDE '13*, pages 1268–1271, 2013.

[4] D. Agrawal, A. El Abbadi, and S. Wang. Secure and privacy-preserving data services in the cloud: A data centric view. *VLDB Endow.*, 5(12):2028–2029, 2012.

[5] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. SIGMOD '04, pages 563–574. ACM, 2004.

[6] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *CIDR '13*, 2013.

[7] A. Arasu, K. Eguro, R. Kaushik, and R. Ramamurthy. Querying encrypted data. SIGMOD '14, pages 1259–1261. ACM, 2014.

[8] S. Bajaj and R. Sion. Trusteddb: A trusted hardware based database with privacy and data confidentiality. SIGMOD '11, pages 205–216. ACM, 2011.

[9] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang. Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward. CCS '15, pages 837–849. ACM, 2015.

[10] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. CRYPTO '11, pages 578–595. Springer-Verlag, 2011.

[11] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious ram practical. Technical report, MIT, 2011. MIT Tech-report: MIT-CSAIL-TR-2011-018.

[12] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS '14*, 2014.

[13] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO '13, Proceedings, Part I*, pages 353–373, 2013.

[14] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *ACNS '05, Proceedings*, pages 442–455, 2005.

[15] Z. Chang, D. Xie, and F. Li. Oblivious RAM: A dissection and experimental evaluation. *PVLDB*, 9(12):1113–1124, 2016.

[16] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *ASIACRYPT '10*, pages 577–594. Springer Berlin Heidelberg, 2010.

[17] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. Controlling data in the cloud: Outsourcing computation without outsourcing control. CCSW '09, pages 85–90. ACM, 2009.

[18] K.-M. Chung, Z. Liu, and R. Pass. Statistically-secure ORAM with $\tilde{O}(\log^2(n))$ overhead. In *ASIACRYPT '14, Proceedings, Part II*, pages 62–81. Springer Berlin Heidelberg, 2014.

[19] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. CCS '06, pages 79–88. ACM, 2006.

[20] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis. Practical private range search revisited. SIGMOD '16, pages 185–198. ACM, 2016.

[21] T. Ge and S. Zdonik. Answering aggregation queries in a secure system model. VLDB '07, pages 519–530. VLDB Endowment, 2007.

[22] C. Gentry. Fully homomorphic encryption using ideal lattices. STOC '09, pages 169–178. ACM, 2009.

[23] E.-J. Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. http://eprint.iacr.org/2003/216/.

[24] O. Goldreich. Towards a theory of software protection. In *CRYPTO '86*, pages 426–439. Springer-Verlag, 1987.

[25] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[26] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious storage with low I/O overhead. *CoRR*, abs/1110.1851, 2011.

[27] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Practical oblivious storage. CODASPY '12, pages 13–24. ACM, 2012.

[28] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. SIGMOD '02, pages 216–227. ACM, 2002.

[29] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu. Secure multidimensional range queries over outsourced data. *The VLDB Journal*, 21(3):333–358, 2012.

[30] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. VLDB '04, pages 720–731. VLDB Endowment, 2004.

[31] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS '12*, 2012.

[32] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. CCS '12, pages 965–976. ACM, 2012.

[33] P. Karras, A. Nikitin, M. Saad, R. Bhatt, D. Antyukhov, and S. Idreos. Adaptive indexing over encrypted numeric data. SIGMOD '16, pages 171–183. ACM, 2016.

[34] J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. EUROCRYPT '08, pages 146–162. Springer-Verlag, 2008.

[35] R. Li, A. X. Liu, A. L. Wang, and B. Bruhadeshwar. Fast range query processing with strong privacy protection for cloud computing. *VLDB Endow.*, 7(14):1953–1964, 2014.

[36] J. R. Lorch, B. Parno, J. Mickens, M. Raykova, and J. Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *USENIX FAST '13*, pages 199–213. USENIX, 2013.

[37] C. Mavroforakis, N. Chenette, A. O'Neill, G. Kollios, and R. Canetti. Modular order-preserving encryption, revisited. SIGMOD '15, pages 763–777, 2015.

[38] National Institute of Standards and Technology. Advanced encryption standard (AES). Federal Information Processing Standards Publications - 197, November 2001.

[39] M. Naveed, M. Prabhakaran, and C. A. Gunter. Dynamic searchable encryption via blind storage. IEEE SP '14, pages 639–654, 2014.

[40] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. SOSP '11, pages 85–100. ACM, 2011.

[41] R. Ramamurthy, R. Kaushik, A. Arasu, and K. Eguro. Querying encrypted data. In *ICDE '13*, pages 1262–1263, 2013.

[42] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *IEEE SP '16*, pages 198–217, 2016.

[43] B. Schneier. Homomorphic encryption breakthrough, 2009. http://www.schneier.com/blog/archives/2009/07/homomorphic_enc.html.

[44] E. Shi, T. H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with o((logn)3) worst-case cost. In *ASIACRYPT '11, Proceedings*, pages 197–214. Springer Berlin Heidelberg, 2011.

[45] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE SP '00*, pages 44–55, 2000.

[46] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE SP '13*, pages 253–267, 2013.

[47] E. Stefanov, E. Shi, and D. X. Song. Towards practical oblivious RAM. In *NDSS '12*, 2012.

[48] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. CCS '13, pages 299–310. ACM, 2013.

[49] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *Proc. VLDB Endow.*, 6(5):289–300, 2013.

[50] P. Williams, R. Sion, and A. Tomescu. Privatefs: A parallel oblivious file system. CCS '12, pages 977–988. ACM, 2012.

[51] Z. Yang, S. Zhong, and R. N. Wright. Privacy-preserving queries on encrypted data. In *ESORICS*, pages 479–495, 2006.

Improving Company Recognition from Unstructured Text by using Dictionaries

Michael Loster, Zhe Zuo, Felix Naumann
Hasso-Plattner-Institute
Potsdam, Germany
firstname.lastname@hpi.de

Oliver Maspfuhl, Dirk Thomas
Commerzbank
Frankfurt am Main, Germany
firstname.lastname@commerzbank.com

ABSTRACT

While named entity recognition is a much addressed research topic, recognizing companies in text is of particular difficulty. Company names are extremely heterogeneous in structure, a given company can be referenced in many different ways, their names include person names, locations, acronyms, numbers, and other unusual tokens. Further, instead of using the official company name, quite different colloquial names are frequently used by the general public.

We present a machine learning (CRF) system that reliably recognizes organizations in German texts. In particular, we construct and employ various dictionaries, regular expressions, text context, and other techniques to improve the results. In our experiments we achieved a precision of 91.11% and a recall of 78.82%, showing significant improvement over related work. Using our system we were able to extract 263,846 company mentions from a corpus of 141,970 newspaper articles.

1. FINDING COMPANIES IN TEXT

Named entity recognition (NER) defines the task of not only recognizing named entities in unstructured texts but also classifying them according to a predefined set of entity types. The NER task was first defined during the MUC-6 conference [8], where the objective was to discover general entity types, such as persons, locations, and organizations as well as time, currency, and percentage expressions in unstructured texts. Subsequent tasks, such as entity disambiguation, question answering, or relationship extraction (RE), rely heavily on the performance of NER systems, which perform as a preprocessing step.

This section highlights the particular difficulties of finding company entities in (German) texts and introduces our industrial use-case, namely risk management based on company-relationship graphs.

1.1 Recognizing company entities

Although there is a large body of work on recognizing entities starting from persons and organizations, to entities like gene mentions or chemical compounds, the current research often neglects the detection of more fine-grained subcategories, such as person roles or commercial companies. In many cases, the "standard" entity classes turn out to be too coarse-grained to be useful in subsequent tasks, such as automatic enterprise valuation, identifying the sentiment towards a particular company, or discovering political and company networks from textual data.

What makes recognizing company names particularly difficult is that in contrast to person names they are immensely heterogeneous in their structure. As such, they can be referenced in a multitude of ways and are often composed of many constituent parts, including person names, locations, and country names, industry sectors, acronyms, numbers, and other tokens, which makes them especially hard to recognize. This heterogeneity is expected to be true particularly for the range of medium-sized to small companies. Regarding examples like "Simon Kucher & Partner Strategy & Marketing Consultants GmbH", "Loni GmbH", or "Klaus Traeger", which all are official names of German companies, one can easily see that they vary not only in length and types of their constituent parts but also in the position where specific name components appear. In the example "Clean-Star GmbH & Co Autowaschanlage Leipzig KG" the legal form "GmbH & Co KG" is interleaved with information about the type of the company (carwash) and location information (Leipzig, a city in Germany). What is more, company names are not required to contain specific constituent parts: the example "Klaus Traeger" from above is simply the name of a person. It does not provide any additional information apart from the name itself, which leads to ambiguous names that are difficult to identify in practice.

Additionally, and in contrast to recognizing named entities from English texts, detecting them in German texts presents itself as an even greater challenge. As pointed out by Faruqui and Padó, this difficulty is due to the high morphological complexity of the German language, making tasks such as lemmatization much harder to solve [5]. Hence, features that are highly effective for English often lose their predictive power for German. Capitalization is a prime example of such a feature. Compared to English, where capitalization of common nouns serves as a useful indicator for named entities, in German *all* nouns are capitalized, which drastically lowers the predictive power of the feature.

We propose and evaluate a named entity recognizer for German company names by training a conditional random field (CRF) classifier [13]. Besides using different features,

©2017, Copyright is with the authors. Published in Proc. 20th International Conference on Extending Database Technology (EDBT), March 21-24, 2017 - Venice, Italy: ISBN 978-3-89318-073-8, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

Series ISSN: 2367-2005

10.5441/002/edbt.2017.82

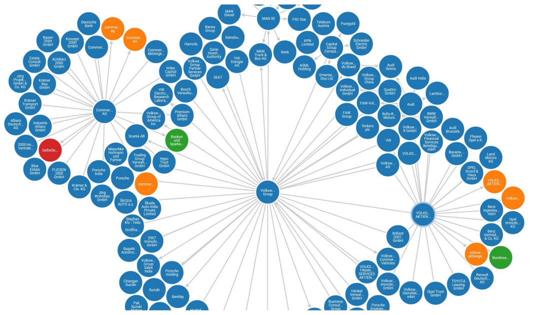


Figure 1: An example of a company graph.

the fundamental idea is to include domain knowledge into the training phase of the CRF by using different real-world company dictionaries. Transforming the dictionaries into token tries enables us to determine efficiently whether the analyzed text contains companies that are included in the dictionary. During a preprocessing step, we use the token trie to mark all companies in the analyzed text that occur in the used trie. In addition, we automatically extend the dictionaries with carefully crafted variants of company names, as we expect them to occur in written text.

1.2 Use case: Risk management using company graphs

Among the many possible applications for a company-focused NER system, we focus on modern risk management in financial institutions as one that would benefit from such a system. Named entity recognition and subsequent relationship extraction from text for the purpose of risk management in financial institutions is particularly important in the context of illiquid risk [1]. Illiquid financial risks basically represent contracts between two individuals, e.g., a bank granting a credit over 1 Mio USD (creditor) to a private company (obligor). Because the risk that the credit-taking company will not honor its repayment obligations cannot be easily transferred to other market participants, assessing the creditworthiness of an obligor is of major importance to the relatively small number of its creditors and other business partners. Also, insights gained by one bank on the obligor's ability to pay back are usually not shared. Hence, obtaining adequate and timely information about non-exchange-listed obligors becomes a difficult task for creditors.

To circumvent this difficulty, financial institutions rely on the so-called "insurance principle": pooling a huge number of independent gains or losses ultimately results in the diversification of risk, which in turn eliminates almost all of it. Unfortunately, risk mitigation based on the insurance principle relies on the independence assumption between individual gains or losses. At the latest with the financial crisis of 2008/2009, this low dependency assumption has turned out to be devastatingly wrong. Information on the economic dependency structure between contracting parties and assets can be seen as the holy grail of financial risk management.

Traditionally, the internal and external data sources used to assess credit risk focus on individual customers, not on the relationships between them. Dependency information is inferred from exposure to common risk factors and thus is inherently symmetric. Direct non-symmetric dependencies, such as supply chains, are not captured.

Fortunately, with the growing amount of openly available data sources, there is justified hope that dependency modeling becomes significantly easier by leveraging this vast amount of data. Sadly, most of those data sources are text-based and require considerable effort to extract the contained knowledge about relationships and dependencies between the entities of interest. The desired outcome of such an extraction effort can be organized in a graph as shown in Figure 1. The figure shows an example of an actual company graph. To be able to automatically extract such graphs from large amounts of unstructured data, a reliable NER system constitutes the first decisive prerequisite for a following relation extraction step.

As pointed out at the beginning, the described use case is merely one of many possible use cases, others might include semantic role labeling, machine translation, and question answering systems.

1.3 Contributions and structure

We address the problem of recognizing company names from textual data by incorporating dictionary matches into the training process using a feature that represents whether a token is part of a known company name. Our evaluation focusses on analyzing the impact of using a perfect dictionary and different real-world dictionaries, as well as the effects of different ways to integrate the knowledge contained in the dictionaries on the performance of the NER system. In particular, we make the following contributions:

- Creation of a NER system capable of successfully recognizing companies in German texts with a precision of 91.11% and a recall of 78.82%.
- Analysis of the impact of various dictionary-based feature strategies on the performance of the NER.

The remainder of this paper is organized as follows: Section 2 discusses related work, while Section 3 presents the baseline configuration for the CRF. In Section 4 we give an overview of the text corpus and the dictionaries we used. We describe the key data structures and technical aspects of the approach in Section 5. Finally, Section 6 presents our experimental results and Section 7 concludes the paper.

2. RELATED WORK

Since its first appearance on the MUC-6 conference [8], the problem of named entity recognition (NER) has become a well-established task leading to many systems and methods that have been developed over time [16]. Before discussing the differences of our approach to the most related approaches, we start by giving an overview of the related work.

Most existing NER systems can be classified into *rule-based* [3, 21], *machine learning-based* [15, 27], or *hybrid* systems [10, 22]. While rule-based systems make use of carefully hand-crafted rules, machine learning approaches tend to train statistical models, such as Hidden Markov Models (HMM) [27] or Conditional Random Fields (CRF) [13], to identify named entities. Hybrid systems combine different methods to compensate their individual shortcomings. They try to incorporate the best parts of the applied methods to reach a high system performance.

Currently, many approaches to the NER problem rely on CRFs [5, 12, 15]. One of the most popular and freely available NER choices for English texts is the Stanford NER system [6]. It recognizes named entities by employing a linear-chain CRF to predict the most likely sequence of named entity labels. While this system shows good performance on English texts, it's performance values decrease when applied to German texts. This effect has also been pointed out by Benikova et al. [2], who argue that German NER systems are not on the same level as their English counterparts despite the fact that German belongs to the group of well-studied languages. This difficulty arises from the fact, that the German language has a very rich morphology, making it especially challenging to identify named entities. Besides the already mentioned problem of capitalization, the German language is capable of creating complex noun compounds like "Vermögensverwaltungsgesellschaft" (asset management company) or "Industrieversicherungsmakler" (industry insurance broker), which make the application of traditional NLP methods even harder.

Nonetheless, German NER systems exist, and some were presented at the CoNLL-2003 Shared Task [23]. With the participating systems achieving F_1 scores between 48% and 73%, the winning system [7] obtained an overall F_1-measure of 72.41% on German texts and 64.62% on recognizing organizational entities. Since the creation of systems for the CoNLL-2003 Shared Task more than ten years ago, one of the most successful NER systems for the German language was introduced by Faruqui and Padó [5]. It reaches overall F_1 scores between 77.2% and 79.8% by using distributional similarity features and the Stanford NER system. Even more recently, additional German NER systems were presented at the GermEval-2014 Shared Task [2]. The GermEval Shared Task specifically focuses on the German language and represents an extension to the CoNLL-2003 Shared Task. The three best competing systems were ExB [9], UKP [19], and MoSTNER [20]. All of them apply machine learning methods, such as CRFs or Neural Nets, which leverage dependencies between the utilized features. Additionally, they use semantic generalization features, such as word embeddings or distributional similarity to alleviate the problem of limited lexical coverage, which, according to [26], is triggered by the often insufficient corpus size used in the training phase of statistical models. To summarize the performance of these systems, they operate in the range of 73% to 79% F_1-measure.

Considering the role of dictionaries in the process of building NER systems, Ratinov and Roth [18] argue that they are crucial for achieving a high system performance. The process of automatically or semi-automatically creating such dictionaries from various information sources has been addressed by [11, 18, 24]. Their research focuses on automatically creating large dictionaries, also known as gazetteers, from open and freely available data sources, such as Wikipedia. The general idea is to establish and assign category labels for each word sequence representing a viable entity by using the information contained in corresponding Wikipedia articles. According to [24], dictionaries can be separated into two different classes, so-called *trigger dictionaries*, which contain keywords that are indicative for a particular type of entity, and *entity dictionaries*, which are comprised of the entire entity labels. For example, a trigger dictionary for companies would most likely contain legal-form words for companies, such as "GmbH" (LLC) or "OHG" (general partnership), whereas an entity dictionary would contain the entire representation of the entity itself, e.g., "BMW Vertriebs GmbH". For our approach we decided to employ entity dictionaries, because there are many openly available data sources from which they can be constructed. Similar to semantic generalization features, features generated from dictionaries aim to mitigate the unseen word problem resulting from the low lexical coverage of statistically learned models.

Many systems make use of dictionaries to increase their performance. All systems mentioned above use dictionaries at some point in their process [9, 19, 20]. Most of the currently existing systems integrate the knowledge contained in dictionaries by constructing features that represent a dictionary lookup. Since each dictionary accounts for a particular type of entity, the constructed feature encodes to which dictionary the word currently under classification belongs and, therefore, implicitly provides evidence for its correct classification. These features are subsequently used in the training process of statistical models, such as CRFs or HMMs.

Another way of integrating dictionary knowledge into the training process of an NER system is described by Cohen and Sarawagi [4]. They present a semi-Markov extraction process capable of classifying entire word sequences instead of single words. By doing so, they effectively bridge the gap between NER methods that sequentially classify words and record linkage metrics that apply similarity measures to compare entire candidate names.

While the previously mentioned systems focus on detecting entities belonging to the entity class "organization", which, apart from companies, includes sports teams, universities, political groups, etc., our system, driven by our use case, specifically excludes such entities and solely focuses on detecting commercial companies. By using a preprocessing step that utilizes external knowledge from dictionaries, we annotate already known companies, which enables us to construct a feature that we use to train a CRF classifier. We concentrate on integrating the knowledge contained in the dictionary into the training process of the classifier. In this way, we use dictionaries from different sources and examine their impact on the overall system performance. Additionally, we report on strategies to integrate the domain knowledge provided by the dictionaries into the training process.

3. CONDITIONAL RANDOM FIELDS AS NER BASELINE

For the construction of our company-focused NER system, we use the CRFSuite Framework[1] to implement a conditional random field model (CRF), as one of the most popular models for building NER systems.

For the baseline configuration of the system, we used various features, such as n-grams, prefixes and suffixes, that are based on those used in the Stanford NER system [6]. Besides regarding different window sizes for each feature, we considered a variety of additional features, for example a token-type feature reducing the type of a token to categories like InitUpper, AllUpper etc., a feature that concatenates different prefix and suffix lengths for each token or features that try to capture some specific characteristic of German company names. However, these features did not result in additional improvements of our baseline configuration. In the end we arrived at a baseline configuration that consists of the following features:

[1] http://www.chokkan.org/software/crfsuite/

	The	auto	maker	VW	AG	is	now...
words :	w_{-3},	w_{-2},	w_{-1},	w_0,	w_1,	w_2,	w_3,
pos-tags :		p_{-2},	p_{-1},	p_0,	p_1,	p_2,	
shape :			s_{-1},	s_0,	s_1		
prefixes :			pr_{-1},	pr_0,			
suffixes :			su_{-1},	su_0,			
n-grams :				n_0,			

Here, the w symbol encodes the word token features of a text with its subscript marking the position of a token. Thus, w_0 refers to the current token whereas w_{-1} and w_1 refer to the previous and next tokens, respectively. The symbols p and s represent the part-of-speech and word shape features with analog subscript notation.

For the creation of POS tags we used the Stanford log-linear part-of-speech tagger [25]. As the name suggests, the shape feature condenses a given word to its shape by substituting each capitalized letter with an X and each lower case letter with an x. Thus the word "**Bosch**" would be transformed to "**Xxxxx**". We also added prefix and suffix features (pr, su) for the current and previous word. These features generate all possible prefixes and suffixes for the specific word. As the last feature we include the set n_0 of all n-grams of the term with n between 1 and the word length of the current word. This feature set yielded the best performance metrics for our baseline configuration without adding any external knowledge besides POS tags.

The baseline system achieves an F_1-measure of 80.65%. More detailed performance metrics of the baseline are presented later in Table 2, in the context of our overall experiments.

4. CORPUS & DICTIONARIES

Before describing our approach in Section 5, we introduce and examine the text corpus and the different information sources we used for building our dictionaries.

4.1 Text corpus

Our evaluation corpus consists of 141,970 documents containing approximately 3.17 million sentences and 54 million tokens. The documents were collected from five German newspaper websites, namely Handelsblatt, Märkische Allgemeine, Hannoversche Allgemeine, Express, and Ostsee-Zeitung. We intentionally selected not only large, national newspapers but also smaller, regional ones; we observe that larger newspapers have a tendency to report more about larger companies or corporations, while the regional press also mentions smaller companies due to their locality in the region. Thus, we expect to increase our chances of discovering smaller and middle tier companies (SMEs) in the long tail by using regional articles in our training process. We extract the main content from the articles by using jsoup[2] and hand-crafted selector patterns, which give us the raw text without HTML markup. Using our final NER system, we were able to extract a total of 263,846 company mentions from this corpus.

4.2 Dictionaries

To build our dictionaries we used two official information sources: the Bundesanzeiger (German Federal Gazette)[3] and the Global Legal Entity Identifier Foundation (GLEIF), which hosts a freely available company dataset[4]. Additionally, we used DBpedia[5] to account for large businesses and the German Yellow Pages[6] to cover middle-tier and local businesses. To simulate a best-case scenario, we composed a "perfect" dictionary containing all manually annotated companies from our testset. Finally, our last dictionary consists of the union of all dictionaries except the perfect one. Although the information sources discussed below contain many different attributes, we use only the company name for the creation of each dictionary.

Bundesanzeiger (BZ). The Bundesanzeiger is the official gazette for announcements made by German federal agencies. Among other things in contains official announcements from companies of various legal forms, such as corporations, limited liability companies, and others including those of foreign companies. Regarding this function, the role of the Bundesanzeiger, as well as the information it provides, is comparable to the U.S. Federal Register. By crawling the BZ company announcements we obtained 793,974 company names, their addresses, and their commercial register ID.

GLEIF (GL). The Global Legal Entity Identifier Foundation (GLEIF) was founded by the International Financial Stability Board[7] in 2014. It is a non-profit organization set up to aid the implementation of the Legal Entity Identifier (LEI). The LEI is designed to be a globally unambiguous, unique identifier for entities that partake in financial transactions. In this context, the dataset of legal entities assigned with a unique LEI is made available for public use by GLEIF. An entry in the provided dataset is, among other data, comprised of the LEI number, legal name, legal form, and address of a legal entity. At the time of writing, the dataset consists of 413,572 legal entities from all global countries that have been assigned a LEI. The subset for German legal entities (**GL.DE**) consists of 42,861 entries.

DBpedia (DBP). The DBpedia project is an effort to systematically extract information from Wikipedia and provide it to the public in a structured way [14]. Structuring the data contained in Wikipedia pages enables us to use query languages like SPARQL to answer complex queries based on data originated from Wikipedia. We queried for the names of all companies contained in the German DBpedia database, yielding a dictionary of 41,724 entries. The resulting dataset contains only companies that have a corresponding Wikipedia page. Thus, we expect that most of the collected company names in this dataset belong to larger, more important companies. Since the extracted names originating from Wikipedia pages, they are very often already in their colloquial form. Also, the dataset contains some additional aliases, such as "VW" for the "Volkswagen AG", which are difficult to generate automatically.

Yellow Pages (YP). As a marketing solutions provider, the German Yellow Pages maintains a large company register, which mainly contains information about small and middle-tier businesses. Using the web pages provided by the register, we were able to extract information, such as the company name, address, email address, phone number, and industrial sector for each company listed in the Yellow

[2] https://jsoup.org

[3] https://www.bundesanzeiger.de

[4] https://www.gleif.org

[5] http://wiki.dbpedia.org

[6] http://www.gelbeseiten.de

[7] http://www.fsb.org/

Pages. The dataset consists of 416,375 company entries.

Perfect Dictionary (PD). For evaluation purposes, we manually labeled company mentions in 1,000 documents (see Sec. 6.1 for details). The perfect dictionary contains exactly the 2,351 manually annotated companies from our training and testset. Because of their origin, the company names contained in this dictionary are already in their colloquial form. Hence, by using this dictionary in our approach, we were indeed able to correctly identify all companies occurring in our testset. Furthermore, this dictionary enables us to simulate the best case scenario in which the dictionary is comprised of all companies occurring in our testset.

All aforementioned dictionaries contain large sets of German company names, so we expect them to overlap. To gain a better understanding of our dictionary's coverages, we computed their mutual containment. We calculated the overlaps using exact match and a fuzzy match. The latter constitutes a more realistic matching scenario accounting for typos and other noise. For computing the matches we applied the method described in [17]. Summarizing their approach, the authors compute the similarity between two strings by splitting them up into n-grams and using similarity measures like Dice, Jaccard, or cosine similarity to determine their similarity using a threshold α. For our calculations we chose a trigram tokenization of the strings and cosine similarity as our metric. We calculated the fuzzy overlaps using different thresholds, and observed that a value of $\theta = 0.8$ performs best on our data.

The pairwise overlaps are shown in Table 1 on the left for exact matches and on the right for fuzzy matches. Surprisingly, even in the case of fuzzy overlaps, the highest overlap was only 11.24%, namely between the BZ and the GL dictionary. All other overlaps were below this value, except in cases where they were contained in each other (GL.DE⊂GL). The exact matching overlaps scored even lower with a maximum overlap of 1.37%.

We identified three possible reasons for these low overlaps. The first and most obvious reason is that our quite simplistic fuzzy matching is not sufficient to recognize many correct matches. Secondly, each of the dictionaries favors a different kind of company names and company sizes. For example, the DBpedia dictionary contains mostly colloquial names whereas the Bundesanzeiger refers to companies using their full legal name. Finally, the dictionaries where crawled at slightly different points in time, hence some of them may contain companies that no longer exist and are thus missing from the other dataset. As a consequence, we created an additional dictionary where we combined all of the mentioned dictionaries into one:

All Dictionaries (ALL). This dictionary is the union of all company names from all other dictionaries. In total it comprises 1,713,272 company names.

5. COMPANY RECOGNITION USING GAZETTEERS

Named entity recognition (NER) is a sequence labeling task that aims to sequentially classify each word in a given text as belonging to a specific class, e.g., person or company. As mentioned, we make use of the CRFSuite Framework to construct our NER system. First, we describe our alias generation process, which extends the given dictionaries, in Section 5.1. Then, Section 5.2 describes how we create the dictionaries and how we efficiently integrate the contained domain knowledge into the training process of the CRF.

5.1 Alias generation

Unfortunately, company names acquired from web sources contain noise, such as country names, legal forms, and other spurious terms. That is, they often differ significantly from their colloquial names. Here the "colloquial name" is to be understood as the name by which a company is commonly referred to in text. For example, while "Dr. Ing. h.c. F. Porsche AG" represents the official company name of the automobile manufacturer, we most often refer to the company by its colloquial name, which is simply "Porsche". Assuming that articles mention companies more frequently by their colloquial name then their official name, it becomes necessary to automatically derive such alternative names, in the following referred to as *aliases*, from a company's official name.

Regarding the alias generation, special attention should be paid to the fact that one company often possesses more then one alias. Considering again the example from above, the company Porsche has at least four valid and common aliases, namely "Dr. Ing. h.c. F. Porsche AG", "Ferdinand Porsche AG", "Porsche AG", or just plain "Porsche". Furthermore, there are a number of non-trivial aliases that are particularly difficult to anticipate by using an automated process. For example the automobile manufacturer "Volkswagen" is also referred to as "VW" or even "die Wolfsburger", referring to the the town of Wolfsburg, in which Volkswagen's headquarters is located.

Our alias generation process consists of the following five steps, using the example of TOYOTA MOTOR$^{\text{TM}}$USA INC..

	Step	Example
1	Removal of legal form designations	TOYOTA MOTOR$^{\text{TM}}$USA
2	Removal of special characters	TOYOTA MOTOR USA
3	Normalization	Toyota Motor USA
4	Country name removal	Toyota Motor
5	Stemming of company names	no change

Each of the Steps 1–4 yields one new alias for the currently processed company name resulting in four aliases per name. Note that some of the four aliases are identical and identical copies are removed. The fifth and final stemming step adds another five aliases by stemming the company name itself and all previously generated aliases. This means that a maximum of nine aliases could be generated by applying the five processing steps to a given company name.

1 & 2: Legal form & special character cleansing. We start to infer the aliases by using a rule-based approach based on regular expressions to strip away a company's legal form. The regular expressions we use are derived from the description of business entity types, found on Wikipedia[8]. The derivation process consists of looking at the business entity types for selected countries and manually creating regular expressions that are able to match the legal forms of the selected countries. We chose the countries based on the most frequent legal forms occurring in our datasets.

[8] http://en.wikipedia.org/wiki/Types_of_business_entity

	Exact match overlaps						Fuzzy match overlaps (cosine, $\theta = 0.8$)					
	BZ	**DBP**	**YP**	**GL**	**GL.DE**	**PD**	**BZ**	**DBP**	**YP**	**GL**	**GL.DE**	**PD**
BZ	796,389	-	-	-	-	-	796,389	4,746	114,958	122,308	119,514	4,900
DBP	333	41,724	-	-	-	-	2,436	41,724	2,049	3,472	1,775	857
YP	14,689	757	416,375	-	-	-	38,170	3,141	416,375	7,988	7,741	330
GL	16,420	792	2,166	413,572	-	-	25,419	4,569	6,546	413,572	43,838	504
GL.DE	16,370	452	2,130	42,861	42,861	-	23,372	1,907	6,128	42,861	42,861	249
PD	62	633	105	50	31	2,351	232	821	207	248	125	2,351

Table 1: Exact and fuzzy match dictionary overlaps. For instance, of 796,389 BZ entries, only 333 find and exact and 2,436 find a similar entry in DBP.

For example, the business entity types we used to derive the regular expressions for Germany include “Gesellschaft bürgerlichen Rechts (GbR)”, “Kommanditgesellschaft (KG)”, or “Offene Handelsgesellschaft (OHG)”.

Step 2 further cleanses the names by removing various special characters, such as “®”, “TM” and parentheses.

3: Normalization of company names. In Step 3, we tokenize the company name and “normalize” each token that has a length greater than four characters and is written in all capital letters. This normalization step consists of first lowercasing and then capitalizing each token that matches the aforementioned criterion. As an example, the normalization step would transform “VOLKSWAGEN AG” into “Volkswagen AG” and “BASF INDIA LIMITED” into “BASF India Limited”.

4: Country name removal. During fourth step we remove all country names appearing in a company’s name using a list of country names and their translations to other languages[9]. Although in general more intricate transformation rules can be created, we found that the ones presented here are sufficient for our purposes.

5: Stemming. Unfortunately, the technique described in the next section, which we employ to verify whether a token sequence is contained in a dictionary has some drawbacks. Using an exact matching strategy to match company names that deviate only slightly from the aliases stored in a dictionary can produce suboptimal results. For example, consider the name “Deutsche Presse Agentur”, which can also occur as “Deutschen Presse Agentur”, depending on the grammatical context. To mitigate these matching issues, we generate additional aliases by stemming each token in a company’s name and all its generated aliases using a German Snowball Stemmer[10]. Using this strategy we generate the alias “Deutsch Press Agentur”, which can in turn be used to match both representations of the aforementioned name. Adding the resulting aliases to a dictionary increases the chances to match a slightly varying company name to an entity contained in the dictionary while using an exact match strategy.

Our experiments shall show that using a stemmed dictionary has only a limited impact on the overall system performance. As the concepts of stemming and lemmatization are closely related, we also expect similar performance using a lemmatized dictionary and thus abstain from lemmatization.

5.2 Dictionary and feature construction

For the creation of the dictionary, we decided to use entity dictionaries, solely containing entire entity names, instead of using trigger dictionaries, which consist mostly of simple keywords. Using this approach simplifies the creation of dictionaries, because we need to add only a given company name to the list, instead of manually creating triggers.

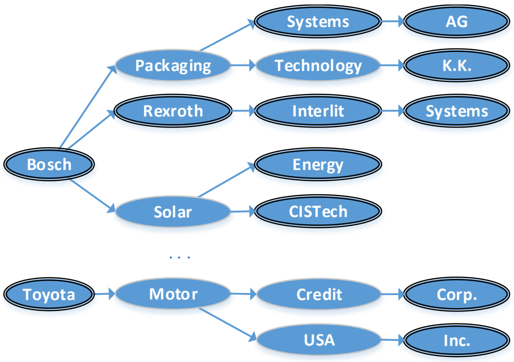


Figure 2: An example of a token trie. Double circles indicate final states.

To make use of the information contained in a dictionary during the CRF training process, we create a feature that encodes whether the currently classified token is part of a company name contained in one of the dictionaries. To efficiently match token sequences in a text against a particular dictionary we tokenize a company’s official name and all its aliases and insert the generated tokens, according to their sequence, into a trie data structure. During the insertion, we mark the last inserted token of each token sequence with a flag, denoting the end of the inserted name. In this manner, we insert all company names into the token trie. Figure 2 shows an excerpt of such a token trie after inserting some company names. After its creation, the token trie functions as a finite state automaton (FSA) for efficiently parsing and annotating token sequences in texts as companies.

We perform the matches in a greedy fashion by always choosing the longest possible match. The outlined approach is crucial when using entity dictionaries. In contrast to trigger dictionaries which contain only single tokens, entity dictionaries mark the entire token sequence representing an entity (e.g., “Volkswagen Financial Services GmbH”) and therefore need to keep track of their matching state to determine if a match occurred.

6. EXPERIMENTS

In this section we describe our experiments and present the results generated by our system. In Section 6.1 we discuss the setup of our experiments by introducing our test data, annotation policy, and the validation method used. Our overall goal is to evaluate the effect of using dictionar-

[9] https://en.wikipedia.org/wiki/List_of_country_names_in_various_languages

[10] http://snowball.tartarus.org/algorithms/german/stemmer.html

ies for NER. Section 6.2 presents the evaluation results of our baseline system *without* the use of dictionaries, as well as a comparative evaluation against the Stanford NER system. The results of using *only* the generated dictionaries to discover companies in our test data are discussed in Section 6.3. Section 6.4 then shows and discusses the results of *integrating* the domain knowledge contained in the dictionaries into our baseline system. Finally, we discuss the case of a *perfect* dictionary in Section 6.5. The performance results in terms of precision, recall and F_1 measure for all analyzed system configurations can be found in Table 2.

6.1 Experimental setup

For the evaluation of our system we randomly selected 1,000 articles for which we could confirm that they contain at least one company mention. We manually annotated these articles by assigning the company-label to each token representing a company mention in the text. We used a very strict annotation policy for tagging the company names in each document; the goal of the policy is to distinguish between mentions referring to a company and mentions referring to related products, persons, or brands. To this end, we considered the context of a company mention to identify a "real" company like BMW, as opposed to a mention appearing as part of another phrase, such as BMW X6, which we did not annotate. In this case, the token X6 identifies the token BMW as part of a product mention. During the annotation process, we discovered and marked 2,351 company mentions in the chosen documents, each consisting of one or more tokens. Links to the news articles of this corpus together with titles and labeled entities are available at https://hpi.de/en/naumann/projects/repeatability/datasets/corpus-comp-ner.html.

To evaluate the performance of our system, we performed a ten-fold cross-validation by splitting the annotated documents into ten folds, each fold containing 900 articles for training and 100 articles for testing. For each fold, we measure precision, recall, and F_1-measure. As usual, the overall performance of the trained model is calculated by averaging the performance metrics over all folds.

We conduct a series of experiments to evaluate our system as well as the impact of different dictionary versions on the systems performance. The results of all experiments are given in Table 2. As our first experiment we compared the performance of our baseline system to the Stanford NER system as described in Section 6.2. Subsequently, we conducted multiple experiments to evaluate the impact of different dictionary versions on the performance of the generated CRF model. Therefore, we generated multiple dictionary versions, which correspond to the rows in Table 2. We created three different dictionary versions for the Bundesanzeiger, GLEIF, GLEIF(DE), Yellow Pages and DBpedia. The first dictionary version contains the original company names obtained from the crawled sources. The second version, marked with "+ Alias", additionally includes all aliases generated by the process described in Section 5.1. The last version, marked with "+ Alias + Stem", also incorporates a stemmed version of each company name and all its generated aliases. We excluded the perfect dictionary from the alias generation process, since it contains the manually tagged colloquial company names. Hence, the approximation of colloquial company names through alias generation is not necessary.

We evaluated each of the generated dictionary versions in two scenarios, illustrated by the two columns "Dict only" and "CRF" in Table 2. In the "Dict only" scenario, described in Section 6.3, we use each dictionary on its own to identify the companies contained in our testset. The "CRF" scenario is discussed in Section 6.4 where we focused on integrating the different dictionary versions into the training process of the CRF and use the generated model to discover company names.

6.2 No dictionaries

We started our experiments by evaluating the baseline configuration introduced in Section 2. Using the basic features mentioned there, we were able to achieve a performance of F_1=80.65% without adding any additional domain knowledge to the system (see Table 2 for details).

We additionally compare our baseline system to the Stanford NER system [6] as one of the most popular NER systems. We used the Stanford system to train a new model on the same training and test documents as for our system, using the configuration suggested on their web-page[11]. Using the resulting model, the Stanford system achieves a slightly better F_1 score of 81.76%. This result is 1.36 percentage points below the precision and 2.68 percentage points above the recall metrics, due to slight variations in the features used.

6.3 Dictionaries only

Next, we used the generated dictionaries on their own to discover the company mentions contained in our testset, as described in Section 5.2. The left, "Dict only" part of Table 2 represents the results of our experiments. The highest precision of 74.23% could be achieved by using the Bundesanzeiger dictionary in its original form. Using the DBpedia dictionary in its original form resulted in the highest F_1-measure value of 51.51%. It is worth noting that using this dictionary in combination with our baseline system and the generated aliases also yielded the best results as described in the following section. Not surprisingly, the highest recall of 72.16% was achieved by combining all dictionaries (except PD) that include the generated aliases and the stemmed name versions.

To understand the impact of alias generation, we compare the average recall of all basic dictionaries, which is 22.92%, with the average recall of all dictionary-extended dictionaries, which is 42.97% (data not shown). The difference of 20,06 percentage points is sufficiently high to justify the use of aliases in principle. Analogously, we analyzed stemming. The average improvement caused by using the dictionaries that include aliases as well as the stemmed names accounted for another increase of 0.21%. However, the improvements of recall are accompanied by an average decrease in precision of 13.46% from the no-aliases to the aliases version, and a further decrease by 14.44 percentage points to a total decrease of −18.28% when including the stemmed versions. In summary, we suggest the use of aliases but refrain from including company name stems in a dictionary.

In addition, we experimented with a dictionary that contained only the company names and their stemmed versions, but no aliases, to assess the impact of stemming on the dictionary-only approach. Here, the precision decreased by 18.94 percentage points while the recall increased only by

[11] http://nlp.stanford.edu/software/crf-faq.shtml

Dictionary	Dict only			CRF		
	P	**R**	**F$_1$**	**P**	**R**	**F$_1$**
Baseline (BL)	–	–	–	91.38%	72.25%	80.65%
Stanford NER	–	–	–	90.02%	74.93%	81.76%
BZ	**74.23**%	3.23%	6.15%	90.90%	75.79%	82.63%
BZ + Alias	16.20%	39.27%	22.91%	91.09%	75.74%	82.63%
BZ + Alias + Stem	6.38%	39.77%	10.98%	90.93%	76.03%	82.78%
GL	34.61%	2.92%	5.37%	90.91%	75.76%	82.62%
GL + Alias	41.71%	50.55%	45.67%	90.78%	77.43%	83.55%
GL + Alias + Stem	18.79%	50.77%	27.39%	90.83%	77.07%	83.36%
GL.DE	68.91%	1.17%	2.29%	90.92%	75.82%	82.66%
GL.DE + Alias	55.78%	21.58%	31.02%	90.97%	76.89%	83.30%
GL.DE + Alias + Stem	39.54%	21.58%	27.85%	90.83%	77.07%	83.36%
YP	16.11%	15.01%	15.53%	91.02%	75.88%	82.73%
YP + Alias	18.34%	21.26%	19.68%	90.92%	75.89%	82.67%
YP + Alias + Stem	7.05%	21.34%	10.58%	90.29%	75.92%	82.72%
DBP	63.13%	43.61%	**51.51**%	**91.25**%	78.54%	84.40%
DBP + Alias	44.18%	53.38%	48.29%	91.11%	**78.82**%	**84.50**%
DBP + Alias + Stem	29.79%	53.47%	38.24%	91.14%	78.76%	84.48%
ALL	20.07%	71.56%	31.33%	90.60%	77.36%	83.43%
ALL + Alias	20.11%	71.80%	31.39%	90.61%	77.33%	83.41%
ALL + Alias + Stem	8.15%	**72.16**%	14.64%	90.94%	76.93%	83.32%
PD (perfect dict.)	81.67%	100.00%	89.90%	94.68%	96.47%	95.56%
PD (perfect dict.) + Stem	81.67%	100.00%	89.90%	94.68%	96.47%	95.56%

Table 2: Results of including different dictionaries into the CRF training process

0.08 percentage points (not shown in Table 2). Hence, we conclude that the stemming of company names has a negative impact on the precision of the dictionary-only approach and does not lead to significant improvement of recall.

When averaging over all the different dictionary versions (without PD) we arrive at an overall performance of 32.39% precision and 36.36% recall. Considering these metrics, it becomes clear that a dictionary-only approach is not sufficient for discovering company names in textual data.

Regarding the perfect dictionary, it is interesting to see that while a recall of 100% could be achieved, the precision reached only a maximum of 81.67%, which is owed to false positives. These are mostly of the form mentioned earlier, where a company name is part of a product name or role description (the VW executive was ...). We expect such errors to be eliminated by the combination with the CRF approach, which makes use of a terms's context.

6.4 Combining dictionaries and CRF

We now discuss the results achieved by combining the domain knowledge contained in the dictionaries and the CRF training process. Overall, we were able to improve the overall performance over the no-dictionary and the dictionary-only approaches, regardless of which dictionary we used. Regarding the right columns of Table 2, we achieved the best results in recall and F_1-measure by using the dictionary generated from the DBpedia including the generated aliases (DBP + Alias) data. Using this dictionary, the system was able to reach an F_1 score of 84.50% with precision and recall values of 91.11% and 78.82%, respectively. By combining the colloquial names already contained in the DBpedia dictionary with the additionally generated alias names, we are able to match more companies than with any of the other dictionaries, explaining our high recall. Interestingly, the initial intuition that combining all dictionaries into one would result in the best performance of our system, turned out not to be true. A more concise dictionary, such as DBpedia, yields the slightly better results.

As we have done in the previous section, we calculated the average change in precision, recall, and F_1-measure. Table 3 shows the average change in performance for gradually evolving our baseline system by including the different dictionary versions. We calculated these values to determine which of the extension steps described in Section 5.1 had the largest impact on system performance. As can be seen, the average change in performance increases significantly moving from the baseline system to a system that uses additional domain knowledge by integrating the basic dictionary version without aliases or stemming. Using additional domain knowledge, the system's precision slightly decreased by 0.45 percentage points, whereas recall and F_1-measure improved on average by 4.28 and 2.43 percentage points, respectively.

Using the dictionary versions containing the generated aliases for each company name, the system gained on average another 0.26 percentage points in F_1-measure. With respect to average precision and recall, the recall increased by 0.49 percentage points while precision slightly decreased by 0.02 percentage points. Due to the alias generation process that condenses a given company name according to the rules described in Section 5.1, we were able to increase the recall while at the same time sustaining precision: we achieved a maximum increase of 6.57 percentage points for recall while the precision decreased only slightly by 0.28% using the DBpedia dictionary including generated alias names. The largest increase of 3.85 percentage points in F_1-measure was also recorded while using the same dictionary. The results suggest that by further improving the alias generation process it should be possible to increase the recall while sustaining high precision.

Regarding dictionaries containing the stemmed version of the original company names and their aliases, we conclude that stemming has only a limited impact; the results produced by including stemmed names are not significantly bet-

Transition	Avg. Precision	Avg. Recall	Avg. F_1
BL $\longrightarrow$ BL + Dict	−0.45%	+4.28%	+2.43%
BL + Dict $\longrightarrow$ BL + Dict + Stem	+0.05%	−0.06%	−0.09%
BL + Dict $\longrightarrow$ BL + Dict + Alias	−0.02%	+0.49%	+0.26%
BL + Dict + Alias $\longrightarrow$ BL + Dict + Alias + Stem	−0.09%	−0.05%	−0.01%

Table 3: Performance change for different dictionary versions, averaged over all dictionaries except PD

ter. For a dictionary version that included only the company names and their stemmed version, the improvements were so low or even negative, that we report only on the average change of using this dictionary in Table 3. As it turned out, the reduction of company names to their stemmed form accounts only for a very limited number of cases. For instance, the airline Lufthansa can be referred to as "Deutsche Lufthansa" or "Deutschen Lufthansa", depending on the grammatical context. By using the common stemmed version ("Deutsch Lufthansa") of these two aliases, it is possible to match both company names. However, such circumstances occur much fewer times for company names than expected.

Because the dictionary feature might add a bias towards labeling known tokens as a company, we also examined how many novel named entities we find, i.e., ones that are not already included in the dictionary. For this experiment, we used each testset in our 10 folds, each consisting of 100 documents not used during the training of the corresponding model. Using the DBpedia including aliases model trained on the remaining 900 documents of each fold, we were able to discover on average 328 company mentions. Examining how many of these company mentions are already contained in the dictionary yielded, that on average 45.85% (≈ 150 companies) of the discovered companies were already included in the dictionary, whereas the remaining 54.15% (≈173) were newly discovered. This shows that although the dictionary feature adds a bias towards already known companies, it is still able to generalize to entities which are not part of the used dictionary.

6.5 Perfect dictionary

To simulate a scenario in which the dictionary can be used on its own to identify the company names in a given text, we use the perfect dictionary. As already mentioned in Section 4, the perfect dictionary consists of all manually annotated company mentions from our test and training sets.

Although using this dictionary yields the highest scores for precision, recall, and F_1-measure, the F_1-measure does not reach 100%. The reason for this behavior can be explained by our strict annotation policy. By using this annotation scheme it becomes hard for the algorithm to avoid producing false positives. Consider the case of recognizing the airline Boeing in the mentions "Boeing" and "Boeing 747". In both cases "Boeing" would be recognized as a company, producing one true positive and one false positive. Hence, a drawback of our system is that the dictionary feature introduces a bias towards companies contained within the dictionary, inducing some false positives if the dictionary feature turns out to be wrong. This problem translates to all other dictionaries that we use. Therefore, we argue that even under ideal circumstances where the dictionary contains all entities that we want to discover, it is not possible to sustain a high precision value by using the dictionary on its own.

Nonetheless, as can be seen by comparing the results in Table 2, using dictionaries to incorporate domain knowledge into the CRF method yields superior results over using them on their own to recognize company names. Considering the average precision, recall, and F_1-measure, the combination of dictionaries and CRF performs significantly better then the pure dictionary approach described in Section 6.3. Integrating the domain knowledge contained in the DBpedia dictionary we achieved a precision of 91.11% and a recall of 78.82%. Regarding the subsequent application or relationship extraction we consider this result as sufficient for recognizing companies in textual data.

7. CONCLUSION & FUTURE WORK

We described a named entity recognition system capable of recognizing companies in textual data with high lexical complexity, achieving a precision of up to 91.11% at a recall of 78.82%. Besides creating the NER system, the particular focus of this work was to analyze the impact of different dictionaries containing company names on the performance of the NER system. Our investigation showed that significant performance improvements can be made by carefully including domain knowledge in the form of dictionaries into the training process of an NER system. On average we were able to increase recall and F_1-measure by 6.57 and 3.85 percentage points, respectively, over our baseline that did not use any external knowledge. Additionally, we showed that applying an alias generation process leads to an increase in recall while sustaining a high precision.

While working with company names, it became increasingly clear that a more sophisticated alias generation process would be needed to handle some of the extremely complex company names. Thus, our future work shall address this issue by including a nested named entity recognition (NNER) step into the preprocessing phase of the dictionary entities. By doing so, we hope to gain semantic knowledge about the constituent parts that form a company name, enabling us to not only increase dictionary quality but to also better determine the *colloquial name* of a company, which in turn would increase the matches of company names in a given text. Another improvement would be to include entities of different entity types (e.g., brands or products) into the token trie, treating them as a blacklist that can then be used to determine whether a sequence of tokens should be marked as a company or not.

The observation that using the smallest dictionary yielded the best results on our newspaper corpus, could indicate that it is important to match the characteristic of the used dictionary with the characteristic of the text corpus. Thus it could be promising to investigate additional corpora, e.g., legal documents, and determine whether dictionaries that are closer to the characteristic of the new corpora also result in a higher system performance.

8. REFERENCES

[1] H. Amini, R. Cont, and A. Minca. Resilience to contagion in financial networks. *Mathematical Finance*, 26(2):329–365, 2016.

[2] D. Benikova, C. Biemann, M. Kisselew, and S. Padó. Germeval 2014 named entity recognition shared task: Companion paper. In *Proceedings of the KONVENS Workshop GermEval Shared Task on Named Entity Recognition*, 2014.

[3] L. Chiticariu, R. Krishnamurthy, Y. Li, F. Reiss, and S. Vaithyanathan. Domain adaptation of rule-based annotators for named-entity recognition tasks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2010.

[4] W. W. Cohen and S. Sarawagi. Exploiting dictionaries in named entity extraction: Combining semi-Markov extraction processes and data integration methods. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2004.

[5] M. Faruqui and S. Padó. Training and evaluating a german named entity recognizer with semantic generalization. In *Proceedings of the Conference on Natural Language Processing (KONVENS)*, 2010.

[6] J. R. Finkel, T. Grenager, and C. D. Manning. Incorporating non-local information into information extraction systems by Gibbs sampling. In *Proceedings of the annual meeting of the Association for Computational Linguistics (ACL)*, 2005.

[7] R. Florian, A. Ittycheriah, H. Jing, and T. Zhang. Named entity recognition through classifier combination. In *Proceedings of the Conference on Natural Language Learning at HLT-NAACL*, 2003.

[8] R. Grishman and B. Sundheim. Message Understanding Conference – 6: A brief history. In *Proceedings of the International Conference on Computational Linguistics (COLING)*, 1996.

[9] C. Hänig, S. Bordag, and S. Thomas. Modular classifier ensemble architecture for named entity recognition on low resource systems. In *Proceedings of the KONVENS Workshop GermEval Shared Task on Named Entity Recognition*, 2014.

[10] M. Hermann, M. Hochleitner, S. Kellner, S. Preissner, and D. Zhekova. Nessy: A hybrid approach to named entity recognition for German. In *Proceedings of the KONVENS Workshop GermEval Shared Task on Named Entity Recognition*, 2014.

[11] J. Kazama and K. Torisawa. Exploiting Wikipedia as external knowledge for named entity recognition. *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, 2007.

[12] V. Krishnan and C. D. Manning. An effective two-stage model for exploiting non-local dependencies in named entity recognition. In *Proceedings of the International Conference on Computational Linguistics (COLING) and Annual Meeting of the Association for Computational Linguistics (ACL)*, 2006.

[13] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. *Proceedings of the International Conference on Machine Learning (ICML)*, 2001.

[14] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. DBpedia – a large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web Journal*, 2014.

[15] A. McCallum and W. Li. Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons. In *Proceedings of the Conference on Natural Language Learning at HLT-NAACL*, 2003.

[16] D. Nadeau and S. Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, 30, 2007.

[17] N. Okazaki and J. Tsujii. Simple and efficient algorithm for approximate dictionary matching. In *Proceedings of the International Conference on Computational Linguistics (COLING)*, 2010.

[18] L. Ratinov and D. Roth. Design challenges and misconceptions in named entity recognition. In *Proceedings of the Conference on Computational Natural Language Learning*, 2009.

[19] N. Reimers, J. Eckle-Kohler, C. Schnober, J. Kim, and I. Gurevych. GermEval-2014: Nested named entity recognition with neural networks. *Proceedings of the KONVENS Workshop GermEval Shared Task on Named Entity Recognition*, 2014.

[20] P. Schüller. MoSTNER: Morphology-aware split-tag german ner with factorie. *Proceedings of the KONVENS Workshop GermEval Shared Task on Named Entity Recognition*, 2014.

[21] S. Sekine and C. Nobata. Definition, dictionaries and tagger for extended named entity hierarchy. *Proceedings of the International Conference on Language Resources and Evaluation (LREC)*, 2004.

[22] R. K. Srihari. A hybrid approach for named entity and sub-type tagging. In *Proceedings of the Applied Natural Language Processing Conference*, 2000.

[23] E. F. Tjong Kim Sang and F. De Meulder. Introduction to the CoNLL-2003 shared task. In *Proceedings of the Conference on Natural Language Learning at HLT-NAACL*, 2003.

[24] A. Toral and R. Muñoz. A proposal to automatically build and maintain gazetteers for named entity recognition by using Wikipedia. *Proceedings of the Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, 2006.

[25] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL)*, 2003.

[26] P. Watrin, L. de Viron, D. Lebailly, M. Constant, and S. Weiser. Named entity recognition for german using conditional random fields and linguistic resources. *Workshop Proceedings of the KONVENS*, 2014.

[27] G. Zhou and J. Su. Named entity recognition using an HMM-based chunk tagger. In *Proceedings of the annual meeting of the Association for Computational Linguistics (ACL)*, 2002.

Temporal group linkage and evolution analysis for census data

Victor Christen
University Leipzig
Germany
christen@informatik.uni-leipzig.de

Anika Groß
University Leipzig
Germany
gross@informatik.uni-leipzig.de

Jeffrey Fisher
Australian National University
Australia
jeffrey.fisher@anu.edu.au

Qing Wang
Australian National University
Australia
qing.wang@anu.edu.au

Peter Christen
Australian National University
Australia
peter.christen@anu.edu.au

Erhard Rahm
University Leipzig
Germany
rahm@informatik.uni-leipzig.de

ABSTRACT

The temporal linkage of census data allows the detailed analysis of population-related changes in an area of interest. It should not only link records about the same person but also support the linkage of groups of related persons such as households. In this paper, we thus propose a new approach to both temporal record and group (household) linkage for census data and study its application for change analysis. The approach utilizes the relationships between individuals to determine the similarity of groups and their members within a graph-based method. The approach is also iterative by first identifying high quality matches that are subsequently extended by matches found with less restrictive similarity criteria. A comprehensive evaluation using historical census data from the UK indicates a high effectiveness of the proposed approach. Furthermore, the linkage enables an insightful analysis of household changes determined by so-called evolution patterns.

1. INTRODUCTION

Census data provides valuable information about individuals and households within cities or regions at a specific point in time [18]. Moreover, the temporal linkage of different census datasets allows analyzing the changes that occur in a population which is of increasing importance for social, demographic, economic and health-related studies [8, 13, 18]. In general, the temporal analysis of changing information about individuals and other entities is seen as a major requirement and challenge for future data analysis [6].

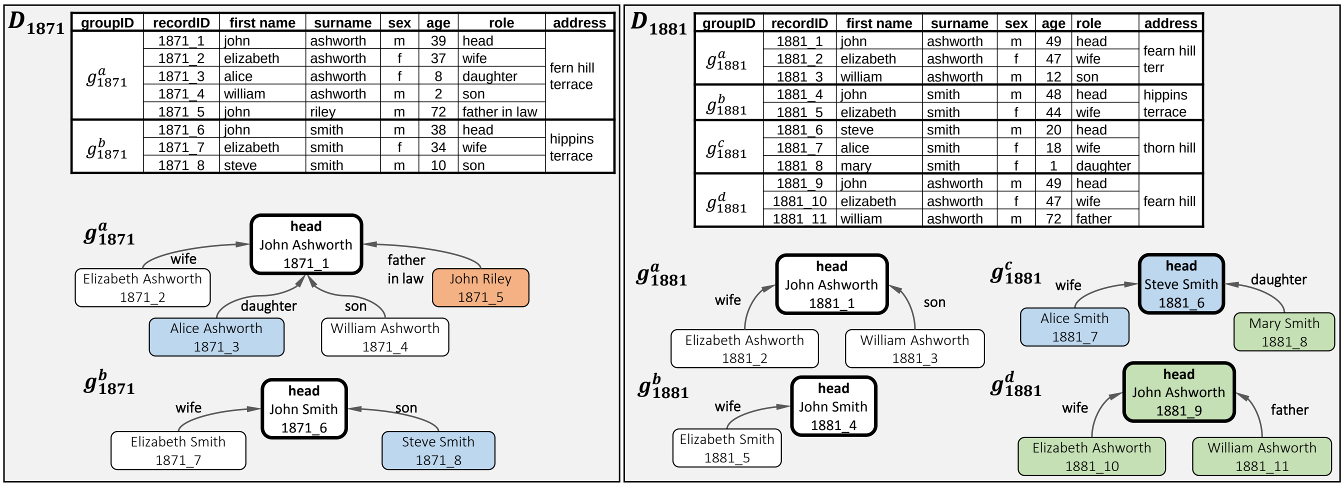
There is a large number of available census datasets for different regions of interest. Normally such census datasets are collected on a regular basis, e.g., every ten years, so that multiple successive versions can be utilized to analyze population- and household-related changes. A key prerequisite for such change studies is the temporal linkage of person records as well as of households, representing a group of individuals living together. There has been a modest amount of previous work on such temporal linkage problems, mainly focusing on temporal record linkage taking into account that linkage-relevant attributes such as surname, address or occupation may change over time [2, 5, 15, 17] (see Section 6). These studies mostly ignore the relationships between individuals, e.g., people living together in a household. Moreover, they do not consider the linkage and evolution of groups of related individuals, such as in a household, which is a main focus of this paper.

Fig. 1 illustrates the problem for two successive historical census datasets from 1871 and 1881. In each dataset, individuals are associated to a single household and have a household-specific relationship or role, such as head of household or daughter (of the head of household). These relationships can be represented in *household graphs* as shown in the lower part of Fig. 1. To understand the changes between the two considered points in time, one has to find matching individuals and their changes which is challenging, in particular due to the occurrence of frequent names (first names like 'John' and 'Elizabeth' or surnames like 'Ashworth' and 'Smith' in our dataset) and attribute changes. Of course, we also need to identify people who occur only in one of the datasets because of deaths, emigration, births and immigration. Obviously, a person in one census dataset should match to at most one person in another census dataset so that temporal linkage aims at a 1:1 mapping between person records. Moreover, we want to identify household-related changes, e.g., to what degree the individuals in a household have stayed together or moved to other households. In this case, we have to identify a many-to-many mapping between households.

In our example in Fig. 1, the daughter of the head of household in g^a_{1871} (*Alice*) married *Steve* from household g^b_{1871} and they both moved into the new household g^c_{1881} as shown in the 1881 census data (see blue nodes in household graphs). *John Riley* died within the considered time

Series ISSN: 2367-2005

10.5441/002/edbt.2017.83



D_{1871}

groupID	recordID	first name	surname	sex	age	role	address
g^a_{1871}	1871_1	john	ashworth	m	39	head	fern hill terrace
	1871_2	elizabeth	ashworth	f	37	wife	
	1871_3	alice	ashworth	f	8	daughter	
	1871_4	william	ashworth	m	2	son	
	1871_5	john	riley	m	72	father in law	
g^b_{1871}	1871_6	john	smith	m	38	head	hippins terrace
	1871_7	elizabeth	smith	f	34	wife	
	1871_8	steve	smith	m	10	son	

D_{1881}

groupID	recordID	first name	surname	sex	age	role	address
g^a_{1881}	1881_1	john	ashworth	m	49	head	fearn hill terr
	1881_2	elizabeth	ashworth	f	47	wife	
	1881_3	william	ashworth	m	12	son	
g^b_{1881}	1881_4	john	smith	m	48	head	hippins terrace
	1881_5	elizabeth	smith	f	44	wife	
g^c_{1881}	1881_6	steve	smith	m	20	head	thorn hill
	1881_7	alice	smith	f	18	wife	
	1881_8	mary	smith	f	1	daughter	
g^d_{1881}	1881_9	john	ashworth	m	49	head	fearn hill
	1881_10	elizabeth	ashworth	f	47	wife	
	1881_11	william	ashworth	m	72	father	

Figure 1: Example census data for two points in time (1871 and 1881). Red / green / blue colored nodes denote individuals who disappear / newly appear / moved to another household.

period (red node for the first census), while the child *Mary Smith* was born (green node for the second census). Furthermore, a new family (household g^d_{1881}) moved into the region. Note that the groups g^a_{1881} and g^d_{1881} have highly similar attribute values, but only g^a_{1871} should be linked to g^a_{1881}. To overcome such ambiguities of person-related attributes, our linkage approach will utilize stable attributes (such as birth year) as well as stable relationships between records, such as family relations or age differences.

In this paper, we propose and evaluate a novel approach for temporal group and record linkage for historical census data that considers the relationships between individuals. Moreover, we use the linked information for an initial change analysis for individuals and households. Specifically, we make the following contributions:

- We propose a new graph-based approach to linking households and person records between successive versions of census data. The approach works in several steps and utilizes an approximate record matching approach to identify pairs of related households. The linkage of households is based on their graph representation, and identifies common subgraphs referring to individuals with stable attributes and relationships. The final record links are derived from the linked subgraphs. The approach is iterative and determines group and record links in multiple rounds with decreasing restrictiveness. In this way we start with finding the best matches and apply less restrictive similarity criteria only for the more difficult to match records and groups.

- We utilize the determined record and group links for an initial change analysis based on different evolution patterns, including the splitting and merging of households.

- We apply and evaluate the proposed approaches for six historical UK census datasets. The evaluation shows that the proposed linkage approaches are highly effective and that they allow insightful observations regarding the changes over time.

In the next section, we formalize our problem of temporal record and group linkage. The linkage approach is described in Section 3, while Section 4 discusses the use of evolution patterns for change analysis. In Section 5, we evaluate our temporal linkage approach and analyze the evolution of households for the considered census datasets. We then discuss related work and conclude.

2. PROBLEM DEFINITION

Our approaches to temporal linkage and evolution analysis work on a set of census datasets $\mathbb{D}$ referring to different points in time. Each dataset D_i of time t_i consists of a set of person records R_i and a set of groups G_i representing households. The records in R_i are homogeneously structured and have attributes such as *first name*, *surname*, *age*, *occupation*, and so on. A group $g_i \in G_i$ consists of associated person records (household members) of R_i as well as relationships between them. Each record is part of one group (household) only, i.e., groups are not overlapping.

Groups are represented as (household) *graphs* g_i=$(V_i$,$E_i)$ where the vertices of V_i correspond to the group members and the edges of E_i represent their relationships. Relationships (edges) have attributes or properties, in particular a relationship type or role, e.g., *daughter*. Such relationships can be part of the input data (as in Fig. 1) or can be derived later, e.g., the age difference between two persons. For our example, we may record in the graph for group g^a_{1871} not only the role *daughter* between *Alice* and her father *John* but also the age difference 31 (39-8). Our algorithm not only determines additional properties such as age differences but also additional relationships among group members, e.g., that $Alice$ and $William$ are siblings with an age difference of 6.

Given these datasets and graphs, we want to determine for each pair $D_i = (R_i, G_i)$ and $D_{i+1} = (R_{i+1}, G_{i+1})$ of successive census datasets a so-called record mapping $\mathcal{M}^{i,i+1}_R$ and a group mapping $\mathcal{M}^{i,i+1}_G$. The *record mapping* $\mathcal{M}^{i,i+1}_R$ includes all pairs of records referring to the same real-world person (person links). The mapping is of cardinality 1:1 since each person in R_i can match with at most one person in R_{i+1} and vice versa:

$$\begin{aligned} \mathcal{M}_R^{i,i+1} : = \{(r_i, r_{i+1}) | (r_i, r_{i+1}) \in R_i \times R_{i+1} \wedge \\ \exists (r_i, r'_{i+1}) \in \mathcal{M}_R \rightarrow r'_{i+1} = r_{i+1} \wedge \\ \exists (r'_i, r_{i+1}) \in \mathcal{M}_R \rightarrow r'_i = r_i \} \end{aligned} \tag{1}$$

A *group mapping* $\mathcal{M}_G^{i,i+1}$ consists of group pairs where a group g_i of G_i corresponds completely or partially to a group g_{i+1} of G_{i+1} according to the common records:

$$\mathcal{M}_G^{i,i+1} : = \{(g_i, g_{i+1}) | (g_i, g_{i+1}) \in G_i \times G_{i+1}\} \tag{2}$$

Group mappings can be of cardinailty many-to-many (N:M) since persons of a household can match persons of several households in a different census.

For our running example of Fig. 1, the record mapping includes seven person links between the white and blue colored graph vertices, e.g. link $(1871\_1, 1888\_1)$ for *John Ashworth* and $(1871\_3, 1888\_7)$ for the link between *Alice Ashworth* and *Alice Smith.* The two groups in the first census dataset are split among two groups each in the second dataset, so that there are four group links including (g^a_{1871}, g^a_{1881}). In our evolution analysis, we will also consider person records and groups that are not reflected in these mappings, e.g. relating to newly occurring or disappeared persons and households.

3. TEMPORAL GROUP LINKAGE

Determining the record and group mappings for the temporal linkage of census datasets is challenging not only due to changing attribute values for the same person (e.g., for surname or occupation) but also due to the high ambiguity and frequent occurrence of certain attribute values, as well as because of data quality issues, e.g., misspelled names, errors for age etc. Group linkage has hardly been studied before [1] and requires a flexible approach to determine many-to-many mappings taking into account that households may split or merge. Similar in spirit to collective entity resolution [1, 20], we determine the similarity between records not only based on attribute values but also considering relationships between records (persons) within a graph-based approach. Furthermore, we not only address record linkage but solve record and group linkage jointly within a combined approach. To better deal with the partially low similarity of matching person records and the need to determine many-to-many group mappings we propose an iterative approach for temporal linkage. We first identify safe matches with a high similarity and then continuously relax the similarity criterion to find additional record and group links.

Algorithm 1 describes our approach for determining a group mapping $\mathcal{M}_G^{i,i+1}$ and a record mapping $\mathcal{M}_R^{i,i+1}$ between two successive census datasets D_i and D_{i+1}. The input of the algorithm includes two similarity functions for record matching and parameters for the iterative adjustment of a similarity threshold δ. We first give a high-level description of the algorithm and its main steps. These steps are then explained in more detail in the four following subsections of this section.

At first, we enrich the graphs for each group (household) in the two input datasets by adding implicit relationships between group members, such as derivable family relations. Moreover, we compute for each relationship between persons the age difference as an additional relationship property for later use in the similarity computations.

The main part of the algorithm is a loop to iteratively identify and extend the group mapping $\mathcal{M}_G^{i,i+1}$ and the record mapping $\mathcal{M}_R^{i,i+1}$. In each iteration, we first apply a similarity function *Sim_func* to determine an initial linking and clustering of person records based on attribute similarities only (pre-matching step). The similarity function *Sim_func* specifies the person attributes, a weighting vector ω, and a similarity threshold δ (i.e., two persons are considered to match if the weighted sum of their attribute similarities exceeds δ). In the first iteration, we apply a high value *δ_high* for δ to start with identifying safely matching persons as a basis for also finding safe group matches. Group matches are only determined for pairs of groups connected by at least one (initial) person link. For such group pairs, we apply a *subgraph matching* to determine shared subgraphs

Algorithm 1: Iterative record and group linkage

```
Input:
-$D_i$: old census dataset
-$D_{i+1}$: new census dataset
-$Sim\_func$: similarity function for initial record matching
-$\Delta$: delta for relaxing similarity threshold
-$\delta\_high$: upper bound of similarity threshold
-$\delta\_low$: lower bound of similarity threshold
-$Sim\_func_{rem}$: similarity function for remaining records
Output:
-$\mathcal{M}_R^{i,i+1}$: record mapping
-$\mathcal{M}_G^{i,i+1}$: group mapping
// initialization
$\mathcal{M}_R^{i,i+1} \leftarrow \emptyset, \mathcal{M}_G^{i,i+1} \leftarrow \emptyset$
$\mathcal{M}_R^{p} \leftarrow \emptyset, \mathcal{M}_G^{p} \leftarrow \emptyset$
$G_i \leftarrow$completeGroups $(G_i)$
$G_{i+1} \leftarrow$completeGroups $(G_{i+1})$
$Sim\_func.\delta \leftarrow \delta\_high$
// iterative subgraph matching
repeat
    // identification of candidates
    $\mathcal{C} \leftarrow$ prematching $(R_i, R_{i+1}, Sim\_func)$
    // subgraph matching and criteria computation
    $Sub_G \leftarrow$subgroups $(\mathcal{C}, G_i, G_{i+1}, Sim\_func)$
    $\mathcal{M}_G^{p} \leftarrow$selectGroupMatches $(Sub_G)$
    // extend group mapping
    $\mathcal{M}_G^{i,i+1} \leftarrow \mathcal{M}_G^{i,i+1} \cup \mathcal{M}_G^{p}$
    // extend record mapping
    $\mathcal{M}_R^{p} \leftarrow$extractRecordMapping $(\mathcal{M}_R^{p}, Sub_G, R_i, R_{i+1})$
    $\mathcal{M}_R^{i,i+1} \leftarrow \mathcal{M}_R^{i,i+1} \cup \mathcal{M}_R^{p}$
    // extract unlinked records and records that are
       related to unlinked records
    $R_i \leftarrow$ nonMatchedRecords $(R_i, \mathcal{M}_R^{i,i+1})$
    $R_{i+1} \leftarrow$ nonMatchedRecords $(R_{i+1}, \mathcal{M}_R^{i,i+1})$
    $Sim\_func.\delta \leftarrow Sim\_func.\delta - \Delta$
until $\mathcal{M}_G^{p} = \emptyset \vee Sim\_func.\delta < \delta\_low$
// match remaining records
$\mathcal{M}_R^{p} \leftarrow$match $(R_i, R_{i+1}, Sim\_func_{rem})$
$\mathcal{M}_R^{i,i+1} \leftarrow \mathcal{M}_R^{i,i+1} \cup \mathcal{M}_R^{p}$
$\mathcal{M}_G^{i,i+1} \leftarrow \mathcal{M}_G^{i,i+1} \cup extractGroupLinks(\mathcal{M}_R^{p}, G_i, G_{i+1})$
return $< \mathcal{M}_R^{i,i+1}, \mathcal{M}_G^{i,i+1} >$
```

[1] We are only aware of one approach for group-based linkage of census data [8] that is non-iterative and less sophisticated regarding the use of relationships. In our evaluation in Section 5, we will compare the results for this scheme with our approach.

with both matching persons and matching relationships. In general, a group of the first census dataset has several candidate group matches in the second dataset so that we select the best group matches considering multiple criteria such as the degree of record and relationship similarity. The matching subgraphs of linked groups are then used to extract the matching records for inclusion into the record mapping (line 10 of Algorithm 1).

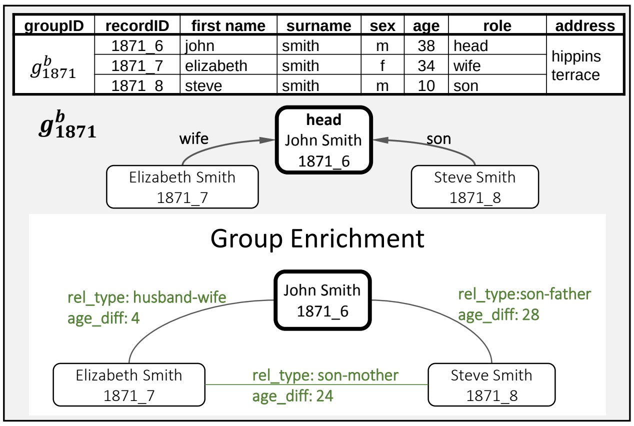
Further iterations only process records not yet included in the record mapping determined so far. We continuously relax the similarity threshold by a decrement Δ until a minimal similarity threshold δ_low is reached (or no further group links are identified). Using such relaxed similarity thresholds aims at finding additional matches between records and groups even in the presence of erroneous or changed attribute values.

After all iterations are performed we have finished subgraph -based group linkage. For the remaining records not yet associated within matching subgraphs, we apply a second attribute-based similarity function Sim_func_{rem} to identify further person links for inclusion into the record mapping (line 17). Moreover, we extend the group mapping by adding the group pairs that are now linked by the newly found record links $\mathcal{M}_G^{i,i+1}$ (line 19).

In the following subsections, we describe the discussed steps in more detail. We start with explaining the preprocessing step to enrich the existing household graphs by implicit relationships and additional relationship properties (Subsection 3.1). In Subsection 3.2, we describe the prematching step of records. In Subsection 3.3, we outline our subgraph matching approach to identify common subgraphs. We then introduce the criteria and algorithm used to select the group matches (Subsection 3.4).

3.1 Group Enrichment

In the initialization phase, we enrich each household group by adding implicit relationships and stable properties such as age differences between persons. In our case, each individual of a household is given a role related to the head of household (which is a special role). This role may not be preserved in future census datasets since individuals may become members of a different household and the head of household may change as well. Hence, comparing households based on these relations only is insufficient in the presence of household changes. We therefore enrich the household graphs by implicit relationships for each record pair of the original group and replace the head-dependent relationship types by a unified type. To increase the semantics of a relationship, we further add the age difference between two household members as a time-independent relationship property. Fig. 2 shows an example of the group enrichment phase for group g^b_{1871}. The relationship between *Elizabeth Smith* and *Steve Smith* is added. Moreover, the age differences age_diff between persons as well as the relationship types rel_type are added to the relationships.



groupID	recordID	first name	surname	sex	age	role	address
g^b_{1871}	1871_6	john	smith	m	38	head	hippins terrace
	1871_7	elizabeth	smith	f	34	wife	
	1871_8	steve	smith	m	10	son	

Figure 2: Example of the group enrichment phase for group g^b_{1871}.

3.2 Pre-Matching

Pre-matching clusters similar records in the census datasets based on their attribute similarity and assigns a cluster label to each record. These labels are utilized to simplify subgraph matching since the labels identify similar records without further similarity computation.

Pre-matching first applies similarity function Sim_func to compare each record of R_i with each record of R_{i+1}. The similarity function specifies the attributes to be compared as well as the attribute-specific similarity function, e.g., q-gram string matching [4]. Furthermore, it uses a weighting vector ω and a required minimum similarity δ. Applying the attribute-specific similarity functions to a pair of records r_i and r_{i+1} results is a similarity vector $\vec{sim}_{(r_i,r_{i+1})}$. Using ω we determine an aggregated similarity $agg\_sim_{(r_i,r_{i+1})}$ by calculating a weighted sum of the attribute similarities:

$$agg\_sim_{(r_i,r_{i+1})} = \omega \cdot \vec{sim}_{(r_i,r_{i+1})} \tag{3}$$

We then keep only the record pairs whose similarity is above the specified threshold δ as potential record matches. Furthermore, we determine the transitive closure or connected components of these match pairs (record links) to cluster together all directly and indirectly matching records. We

Cluster label	recordID	first name	surname
A	1871_1	john	ashworth
	1881_1	john	ashworth
	1881_9	john	ashworth
B	1871_2	elizabeth	ashworth
	1881_2	elizabeth	ashworth
	1881_10	elizabeth	ashworth
C	1871_4	william	ashworth
	1881_3	william	ashworth
	1881_11	william	ashworth
D	1871_6	john	smith
	1881_4	john	smith
E	1871_7	elizabeth	smith
	1881_5	elizabeth	smith
F	1871_8	steve	smith
	1881_6	steve	smith
G	1881_8	mary	smith
H	1871_5	john	riley
I	1871_3	alice	ashworth
K	1881_7	alice	smith

Figure 3: Pre-matching result for running example. Records with the same cluster label represent similar records.

assign to each record of a cluster a unique label, so that records of the same cluster have the same label.

Fig.3 shows the resulting clusters for the running example by using the attributes *first name* and *surname*, $\omega = (0.5, 0.5)$ and similarity threshold 1. Pre-matching results in the shown ten clusters where all records of a cluster share the same first name and surname. We then assign the cluster labels A, B etc. to the respective records of the clusters.

3.3 Subgraph Matching

Subgraph matching looks for common subgraphs in each pair of groups g_i and g_{i+1} of $G_i \times G_{i+1}$ to determine likely group links. To avoid the computation of the cross product between G_i and G_{i+1}, subgraph matching is only applied for pairs of groups sharing at least one similar record, i.e., having the same cluster label.

The subgraph g_{sub} between two groups g_i and g_{i+1} (represented by their enriched graphs with g_i=(V_i,E_i) and g_{i+1}= (V_{i+1},E_{i+1}) consists of a set of vertices R_{sub} and a set of edges E_{sub}. Each vertex in R_{sub} represents a pair of equally labeled (i.e., similar) records v_i from V_i and v_{i+1} from V_{i+1}. Two vertices $(v1_i, v1_{i+1})$ and $(v2_i, v2_{i+1})$ of R_{sub} are connected by an edge of E_{sub} if both the old records $v1_i$, $v2_i$ and the new records $v1_{i+1}, v2_{i+1}$ of these vertices are connected within their enriched graphs of g_i and g_{i+1}, respectively. Furthermore, we require that these edges must have the same relationship type and highly similar relationship properties, in our case regarding the age differences.

Fig. 4 illustrates subgraph matching for group g^a_{1871} from the first census dataset and the two groups g^a_{1881} and g^d_{1881} from the second dataset. For the group pair (g^a_{1871}, g^a_{1881}) we have three matching vertices with labels A, B and C. The three edges have the same relationship types and the same or very similar age differences. The second group pair (g^a_{1871}, g^d_{1881}) also shares three vertices with labels A, B and C but only one of the edges has the same relationship type and similar age difference. Hence the common subgraph is reduced to the one shown in the bottom right of Fig.4.

3.4 Selection of Group Links

Subgraph matching generates candidates for group linkage based on common subgraphs for different group pairs. There may be several linkage candidates per group in G_i and in G_{i+1} so that we have to find the best matching group pairs. The necessary selection should especially guarantee that each record of a group is only linked to one record of another group (This is not the case for the example in Fig.4 where we have two linkage candidates for members of group g^a_{1871}). However, a group can link to more than one group if their subgroups are disjoint.

To select for a certain group g_i the best-matching groups in G_{i+1} we consider all subgraphs g_{sub}=(R_{sub},E_{sub}) involving g_i and apply an aggregated similarity measure. This measure combines three scores capturing the record similarity (Eq. 5), edge similarity (Eq. 6) and the uniqueness (Eq. 7) of a subgroup g_{sub}. The results of the similarity functions are aggregated according to Eq. 4 whereby α determines the influence of record similarity and β represents the weight of edge similarity.

$$g\_sim = \alpha \cdot avg\_sim + \beta \cdot e\_sim + (1 - \alpha - \beta) \cdot unique \tag{4}$$

- *Average Record Similarity*

  For this score we determine the average of the aggregated similarities *agg_sim* for the record pairs of R_{sub}. These aggregated similarities are already determined during pre-matching for each record pair (see section 3.2) and can be obtained from the respective clusters in $\mathcal{C}$.

$$avg\_sim(g_i, g_{i+1}, g_{sub}) = \frac{\sum\limits_{(r_i, r_{i+1}) \in R_{sub}} agg\_sim_{(r_i, r_{i+1})}}{|R_{sub}|} \tag{5}$$

- *Edge Similarity*

  The edge similarity *e_sim* evaluates the similarity of the relationship properties *rp_sim* in the edges in a subgraph, for example the similarity of the age differences between two individuals in the older group g_i vs. the age difference in the newer group g_{i+1}. Furthermore, we apply an aggregation measure similar to the Dice-Coefficient to relate the edge similarities to the total number of relationships of the considered groups g_i and g_{i+1} thereby giving higher weight to those subgraphs covering a large portion of their relationships.

$$e\_sim(g_i, g_{i+1}, g_{sub}) = 2 \cdot \frac{\sum\limits_{e \in E_{sub}} rp\_sim(oldEdge(e), newEdge(e))}{|E_i| + |E_{i+1}|} \tag{6}$$

- *Uniqueness*

  If two group pairs are similar w.r.t both the average record similarity as well as the edge similarity, we like to prefer the group link between the two groups containing records that are less ambiguous than the records of other group pairs. Therefore, we define the uniqueness for a group pair based on the number of vertices of R_{sub} of g_{sub} and the aggregated number of records that are assigned to the same label like the records of R_{sub}. The uniqueness is defined as follows:

$$unique(g_i, g_{i+1}, g_{sub}) = 2 \cdot \frac{|R_{sub}|}{\sum\limits_{r_i \in R_{sub}} |label(r_i)|} \tag{7}$$

  The uniqueness of a group pair g_i and g_{i+1} is 1, if the labels are only assigned to the common records of g_i and g_{i+1} and there exists no other record of R_i or R_{i+1} that has the same label.

For the example of Fig. 4, we obtain the following similarity values for the group pairs (g^a_{1871}, g^a_{1881}) and (g^a_{1871}, g^d_{1881}):

$$\begin{aligned} avg\_sim(g^a_{1871}, g^a_{1881}, g_{sub}) &= \frac{1+1+1}{3} = 1 \\ e\_sim(g^a_{1871}, g^a_{1881}, g_{sub}) &= 2 \cdot \frac{1+1+1}{10+3} = 0.46 \\ unique(g^a_{1871}, g^a_{1881}, g_{sub}) &= 2 \cdot \frac{3}{3+3+3} = 0.66 \end{aligned} \tag{8}$$

$$\begin{aligned} avg\_sim(g^a_{1871}, g^d_{1881}, g_{sub}) &= \frac{1+1}{2} = 1 \\ e\_sim(g^a_{1871}, g^d_{1881}, g_{sub}) &= 2 \cdot \frac{1}{10+3} = 0.15 \\ unique(g^a_{1871}, g^d_{1881}, g_{sub}) &= 2 \cdot \frac{2}{3+3} = 0.66 \end{aligned}$$

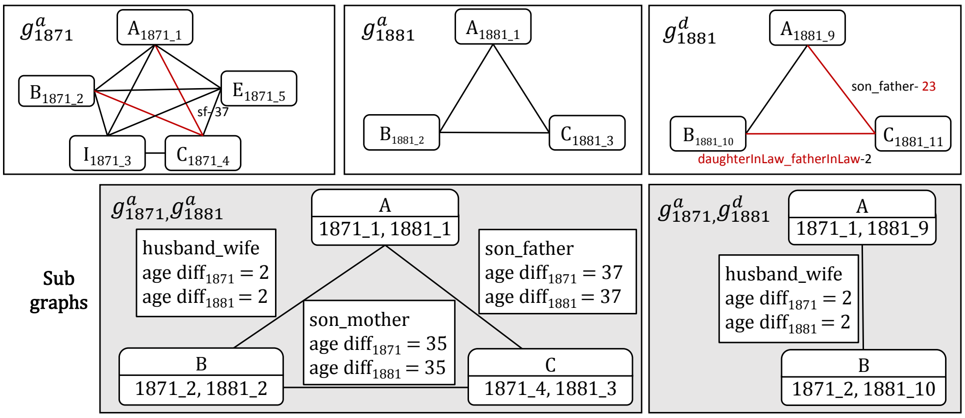


Figure 4: Subgraphs for group pairs (g^a_{1871}, g^b_{1881}) and (g^a_{1871}, g^d_{1881}) of the running example. For (g^a_{1871}, g^d_{1881}), the red-coloured edges are not matched due to a different relationship type or non-similar age difference.

```
Algorithm 2: Selection of group links
Input:
-$Sub_G$: set of quadruples of $<g_i, g_{i+1}, g_{sub}, g\_sim>$
Output:
-$\mathcal{M}^p_G$: partial group mapping
1  $\mathcal{M}^p_G \leftarrow \emptyset$
2  $lookup \leftarrow \emptyset$
   // initialize priority queue ordered by $g\_sim$
3  for $(g_i, g_{i+1}, g_{sub}, g\_sim) \in Sub_G$ do
4      $pq \leftarrow pq.insert(g_i, g_{i+1}, g_{sub}, g\_sim)$
5  while $pq \neq \emptyset$ do
6      $< g_i, g_{i+1}, g_{sub}, g\_sim > \leftarrow pq.max()$
7      $pq \leftarrow pq.remove()$
       // sets of linked records of $g_i$ and $g_{i+1}$
8      $linked\_R_i \leftarrow lookup.get(g_i)$
9      $linked\_R_{i+1} \leftarrow lookup.get(g_{i+1})$
       // records of $g_i$ and $g_{i+1}$ contained in $g_{sub}$
10     $R^i_{sub} \leftarrow getOldRecords(g_{sub})$
11     $R^{i+1}_{sub} \leftarrow getNewRecords(g_{sub})$
12     if $linked\_R_i \cap R^i_{sub} = \emptyset \wedge linked\_R_{i+1} \cap R^{i+1}_{sub} = \emptyset$ then
13         $\mathcal{M}^p_G \leftarrow \mathcal{M}^p_G \cup \{(g_i, g_{i+1})\}$
14         $linked\_R_i \leftarrow linked\_R_i \cup R^i_{sub}$
15         $linked\_R_{i+1} \leftarrow linked\_R_{i+1} \cup R^{i+1}_{sub}$
16         $lookup \leftarrow lookup.update(g_i, processed\_R_i)$
17         $lookup \leftarrow lookup.update(g_{i+1}, processed\_R_{i+1})$
18 return $\mathcal{M}^p_G$
```

The aggregated similarity of these values reaches a higher value for group pair (g^a_{1871}, g^a_{1881}) than for (g^a_{1871}, g^d_{1881}) due to the higher edge similarity of the former pair. As a result, we would only include group pair (g^a_{1871}, g^a_{1881}) in the group mapping and derive the record mapping only for the common subgraph of this pair.

After the determination of the introduced similarity values per subgroup, we apply Algorithm 2 for the selection of the best-matching group pairs. The algorithm follows a greedy strategy by considering subgraphs in the order of their aggregated similarity score. It also considers the disjointness of subgraphs and can determine group mappings of cardinality N:M.

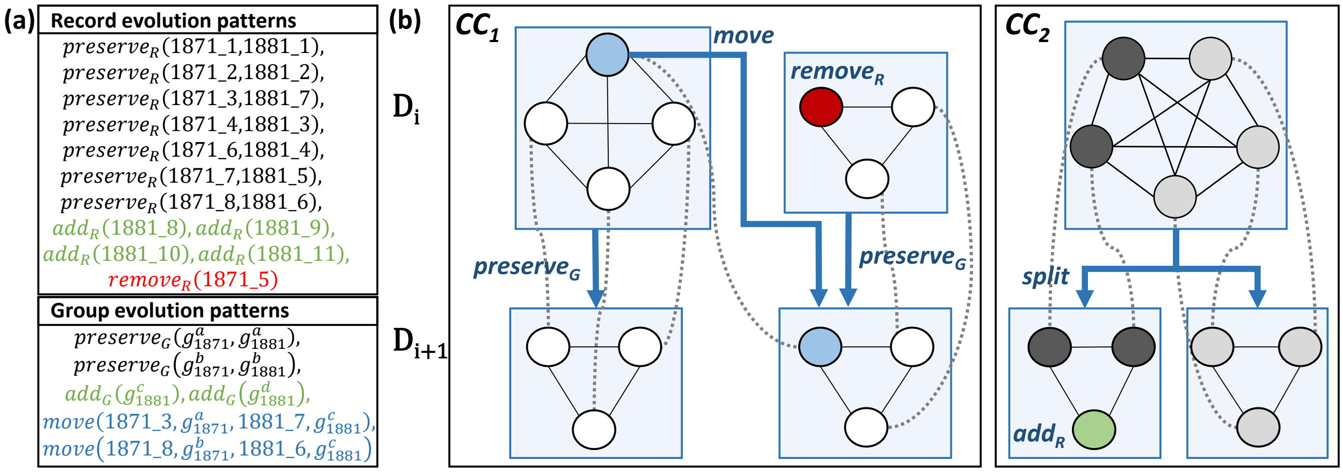
In each iteration, we select the group pair with the highest group similarity from a priority queue *pq*. The selected pair (g_i, g_{i+1}) is added to the group mapping $\mathcal{M}^p_G$ if the overlap between the already linked records of g_i as well as g_{i+1} and the records of the record pairs of g_{sub} is empty (line 12). Thus, we ensure that a record is linked at most to one record. The linked records are represented by $linked\_R_i$ resp. $linked\_R_{i+1}$. Moreover, the records of g_i and g_{i+1} that correspond to a record pair of R_{sub} of g_{sub} are represented by the sets R^i_{sub} and R^{i+1}_{sub}. These sets are returned by *getOldRecords* and *getNewRecords* respectively for a certain subgroup g_{sub}. If a group link is added, we update sets of linked records $linked\_R_i$ and $linked\_R_{i+1}$ for g_i resp. g_{i+1} (line 14 to 17).

Based on the selected group matches, we are able to identify the record matches contained in the corresponding subgraph g_{sub}. The record links are included in each vertex of g_{sub} since R_{sub} is defined as a set of pairs r_i and r_{i+1}. These pairs are the most appropriate links since the related groups are linked.

4. EVOLUTION ANALYSIS

We will now use the results of the temporal record and group linkage to detect changes between different census datasets in order to support the comprehensive evolution analysis of temporal census data. Such a change analysis should not be restricted to a low-level evaluation of individual links but should be realized at a higher, application-specific level to generate relevant and expressive change patterns. We will also include disappearing as well as newly appearing records and groups that are not reflected in the identified mappings but appear only in one of the census datasets. The analysis should further not be limited to two datasets but involve a series of successive census datasets covering longer periods of time.

In this initial study, we use the given census datasets and the determined linkage results to identify a set of basic and more complex changes for records and groups of records that can be identified with the help of so-called evolution patterns (Subsection 4.1). Furthermore, we propose the use of a so-called evolution graph (Subsection 4.2) to provide an aggregated change representation that is extensible to more than two census datasets. Such an evolution graph is a promising basis for advanced graph mining techniques, e.g., to determine frequent or unusual change scenarios.


(a)

Record evolution patterns

$preserve_R$(1871_1,1881_1),
$preserve_R$(1871_2,1881_2),
$preserve_R$(1871_3,1881_7),
$preserve_R$(1871_4,1881_3),
$preserve_R$(1871_6,1881_4),
$preserve_R$(1871_7,1881_5),
$preserve_R$(1871_8,1881_6),
add_R(1881_8), add_R(1881_9),
add_R(1881_10), add_R(1881_11),
$remove_R$(1871_5)

Group evolution patterns

$preserve_G(g^a_{1871}, g^a_{1881})$,
$preserve_G(g^b_{1871}, g^b_{1881})$,
$add_G(g^c_{1881})$, $add_G(g^d_{1881})$,
$move$(1871_3, g^a_{1871}, 1881_7, g^c_{1881}),
$move$(1871_8, g^b_{1871}, 1881_6, g^c_{1881})

Figure 5: (a) Record and group evolution patterns for the running example. (b) Evolution graph and patterns for two successive census datasets D_i and D_{i+1}. Gray dotted lines represent record links, blue arrows indicate evolution patterns between related households.

4.1 Evolution Patterns

We define evolution patterns on individual records and on groups of records. There are three *record evolution patterns* called $preserve_R$, $remove_R$ and add_R. We identify these patterns by utilizing the record mapping $M_R^{i,i+1}$ as well as record sets R_i and R_{i+1} for two successive census datasets D_i and D_{i+1} as follows:

- $preserve_R$ is a record pair representing one individual linked between R_i and R_{i+1}.
  $\forall r_i, r_{i+1} \in R_i \times R_{i+1}$:
  $preserve_R(r_i, r_{i+1}) \leftrightarrow \exists (r_i, r_{i+1}) \in \mathcal{M}_R^{i,i+1}$

- add_R denotes an individual $r_{i+1} \in R_{i+1}$ that is not linked to any record of R_i.
  $\forall r_{i+1} \in R_{i+1} : add_R(r_{i+1}) \leftrightarrow \nexists (r_i, r_{i+1}) \in \mathcal{M}_R^{i,i+1}$

- $remove_R$ denotes an individual $r_i \in D_i$ that is not linked to any record of D_{i+1}.
  $\forall r_i \in R_i : remove_R(r_i) \leftrightarrow \nexists (r_i, r_{i+1}) \in \mathcal{M}_R^{i,i+1}$

To analyze the dynamics of groups, we further define *group evolution patterns* based on changes within groups. These patterns are add_G and $remove_G$ as well as the more complex patterns $preserve_G$, *move*, *split* and *merge*. The patterns $preserve_G$ and *move* both relate to pairs of linked groups but differ on whether the linked groups contain at least two preserved members ($preserve_G$) or only one (*move*). Each pattern is identified by utilizing the census datasets, the group mapping $\mathcal{M}_G^{i,i+1}$ and the record mapping $\mathcal{M}_R^{i,i+1}$:

- add_G denotes a new group $g_{i+1} \in G_{i+1}$ that did not exist in D_i. Thus, the group mapping $\mathcal{M}_G^{i,i+1}$ does not contain any link with g_{i+1}.

- Similarly, $remove_G$ contains a group of $g_i \in G_i$ that does not exist in G_{i+1} anymore.

- $preserve_G$ is a group pair connected by a 1:1 link. Moreover, each group consists of at least 2 individuals satisfying the $preserved_R$ pattern. This condition allows us to identify preserving households across censuses. The requirement that a 'preserved' household should have at least two remaining members is influenced by real-world situations such as households where only the parents remain after their children have moved to another household.

- *move* identifies pairs of linked groups with only one member in common (determined by the $preserve_R$ pattern) that has moved from the old to the new group (household).

- *split* identifies a change situation between a group $g_i \in D_i$ from the old dataset and a set of groups g^a_{i+1}, $g^b_{i+1}, ..., g^k_{i+1} \in G_{i+1}$ in the new dataset, where at least two individuals of g_i must overlap with each of the groups from G_{i+1}. Note, that each individual record can only be contained in one group, i.e., $g^a_{i+1}, g^b_{i+1}, ...,$ g^k_{i+1} are disjoint.

- *merge* covers the opposite situation between a set of groups g^a_i, $g^b_i, ..., g^k_i \in G_i$ from the old dataset and one group $g_{i+1} \in G_{i+1}$ from the new dataset, where at least two individuals from groups in G_i must overlap with the merged group g_{i+1}. Each individual record can only be contained in one group, i.e., $g^a_i, g^b_i, ..., g^k_i$ are disjoint.

Fig. 5(a) shows the corresponding record and group evolution patterns for our running example from Fig. 1. Seven records have been preserved from D_{1871} to D_{1881}. Moreover, there are 4 record additions and one removal. According to the defined group evolution patterns, two groups have been preserved (g^a and g^b), two groups newly appeared in 1881 (add_G for g^c and g^d) and two persons, Alice (1871_3) and Steve (1871_8), moved from their parents' households (g^a_{1871} and g^b_{1871}) to their own new household g^c_{1881}.

4.2 Evolution Graph

Based on the evolution patterns we want to realize further comprehensive evolution analyses for dynamically changing family structures and individual person histories. We propose the use of a so-called *evolution graph* reflecting the history of households across two or more successive census datasets. The graph $\mathcal{G}\_Evolution$ captures both the records and groups per census dataset as vertices and interconnects them across successive datasets by edges that are typed according to the identified evolution patterns (change types). Fig. 5(b) shows a sample evolution graph and evolution patterns for two successive versions D_i and D_{i+1}. Blue boxes represent group vertices and blue arrows represent group evolution patterns, i.e., the changes between households. Two groups have been preserved and are linked via the group pattern $preserve_G$ and one household has been split into two households. One individual moved between two households that are thus connected in the evolution graph. The figure also shows the mapping between individual records (gray dotted lines) as well as a new (add_R) and a removed ($remove_R$) record without incoming/outgoing edges.

The evolution graph enables the application of several graph mining approaches such as cluster analysis, pattern matching or finding frequent subgraphs. One analysis might be to identify households that are preserved across several census periods. A second use case is to identify clusters of related households that can be used for studies of genetic diseases. In Fig. 5(b), a simple computation of connected components on the exemplary evolution graph for two points in time leads to two components consisting of 4 (CC_1) and 3 (CC_2) households, respectively. Running such a computation for larger households graphs for many successive versions can produce longer chains of connected households, e.g., indicating relationships between many generations of families.

5. EVALUATION

In this section, we evaluate the introduced approaches for temporal record and group linkage for different historical census datasets from the UK that have also been used in a previous study [8]. We first describe these datasets and the evaluation setup in Subsection 5.1. We then evaluate the linkage quality of the new approaches for different configurations (Subsection 5.2). In Subsection 5.3 we compare our approach with the results of the previous study [8] as well as with the collective record linkage approach [14]. Finally, we discuss results of an initial evolution analysis for the considered census datasets.

5.1 Datasets and Setup

In our evaluation, we use six census datasets collected from 1851 to 1901 in ten-year intervals from the district of Rawtenstall in North-East Lancashire in the United Kingdom. Table 1 shows an overview of these datasets according to the number of records and households for the different time periods. The table also shows the number of unique value combinations of the first name and surname attributes to illustrate the degree of ambiguity for these attributes. Furthermore, we report the ratio of missing attribute values. The table shows that the number of households and persons has almost doubled within the 50 years period indicating a substantial population growth. There is a high degree of name ambiguity since each combination of first name and surname is far from unique but has an average frequency of up to 2.23 (for 1851) with a highly skewed frequency distribution due to the presence of frequent surnames such as *Ashworth* and *Smith*. Up to 6.5% of the attribute values are missing, which leads to in additional difficulties for finding correct temporal links.

t_i	1851	1861	1871	1881	1891	1901
$\|R_{t_i}\|$	17033	22429	26229	29051	30087	31059
$\|G_{t_i}\|$	3298	4570	5576	6025	6378	6842
$\|fn + sn\|$	7652	10198	13198	15505	17130	19910
$ratio_{mv}$	4.67%	4.19%	3.03%	4.09%	6.33%	6.51%

Table 1: Overview of the census datasets according to the number of records, households, unique combinations of first name and surname $|fn+sn|$ and the ratio of missing values $ratio_{mv}$.

To evaluate the quality of the group and record mappings in terms of precision, recall and F-measure [4], we use the reference mapping determined in [8]. It covers a subset of 1250 matching households from the 1871 and 1881 datasets that consist of 6864 and 6851 members resp. These household were manually linked by experts by focusing on person records found in both datasets.

In our evaluation, we compare different settings for the similarity function considering the string similarity for five attributes and different weight vectors ω_1 and ω_2 as shown in Table 2. We also evaluate different similarity thresholds for pre-matching as well as different weights for determining the aggregated group similarity for selecting group links.

Attribute	Matching method	ω_1	ω_2
First name	q-gram	0.2	0.4
Sex	exact	0.2	0.2
Surname	q-gram	0.2	0.2
Address	q-gram	0.2	0.1
Occupation	q-gram	0.2	0.1

Table 2: Compared set of attributes and the corresponding weighting vector ω to identify the set of blocks $\mathcal{B}$ that are used for the subgraph matching.

5.2 Linkage Evaluation

We first analyze the influence of different similarity functions during pre-matching and then discuss the impact of different similarity functions for selecting matching group pairs. Afterwards we study the effectiveness of incremental linkage.

5.2.1 Influence of pre-matching configuration

The proposed linkage approach builds on the initial record matching and clustering performed in the pre-matching step. We thus start our analysis by comparing the results for determining the attribute similarities based on the two weighting schemes ω_1 and ω_2 (Table 2) and different lower similarity threshold bounds δ_low. For iterative matching we use a start value $\delta\_high = 0.7$ for the similarity threshold δ and $\Delta = 0.05$ for decrementing the threshold until the minimal value δ_low is reached.

Table 3 shows the resulting group and record mapping quality in terms of precision, recall and F-measure for the two weighting schemes and four values of δ_low ranging from 0.4 to 0.55. We observe for all configurations high F-Measure results between 94% and 96% for both the determined record mappings and the group mappings, indicating a very high effectiveness of the proposed approach. The best

parameter	ω	ω_1				ω_2			
	δ_low	0.4	0.45	0.5	0.55	0.4	0.45	0.5	0.55
group mapping	Precision (%)	96.1	96.5	96.7	97.0	97.1	97.1	**97.3**	**97.3**
	Recall (%)	92.2	92.2	92.0	91.7	94.8	**94.8**	**94.8**	94.6
	F-measure (%)	94.1	94.3	94.3	94.2	96.0	96.0	**96.0**	95.9
record mapping	Precision (%)	96.6	96.8	96.8	96.8	97.5	97.5	**97.5**	**97.5**
	Recall (%)	91.9	91.9	91.9	91.8	93.7	93.7	**93.7**	93.7
	F-Measure (%)	94.2	94.3	94.3	94.3	95.6	95.6	**95.6**	95.5

Table 3: Quality of group and record mappings for different weighting vectors ω and lower bounds δ_low.

parameter	(α, β)	(1.0,0.0)	(0.0,1.0)	(0.5,0.5)	(0.33,0.33)	(0.2,0.7)
group mapping	Precision (%)	92.3	96.7	96.6	96.7	**97.3**
	Recall (%)	89.1	94.1	94.3	94.4	**94.8**
	F-Measure (%)	90.7	95.4	95.5	96.0	**96.0**
record mapping	Precision (%)	96.2	97.4	97.3	97.3	**97.5**
	Recall (%)	89.8	93.4	93.4	93.4	**93.7**
	F-Measure (%)	92.9	95.4	95.3	95.3	**95.6**

Table 4: Quality of the group and record mappings for different weights α and β to select matching groups.

F-measure results are generally achieved for $\delta\_low = 0.5$, although the differences are small for the other choices. The simple weighting scheme ω_1 giving equal weight to each of the five considered attributes is consistently outperformed by the alternate approach giving higher weight to attribute *first name* and only reduced weight for the less stable attributes *address* and *occupation*. Pre-matching with weight vector ω_2 thus improves F-measure by around 1.7% for the group mapping and up to around 1.3% for the record mapping.

Of course, there are many more possibilities to define the similarity function and we could also apply learning-based methods to find a near-optimal weight vector [4]. Still our results show that using the similarity function with weight vector ω_2 and $\delta\_low = 0.5$ achieve good and stable results making it an effective default configuration.

5.2.2 Similarity weights for selecting matching groups

We now evaluate the influence of the different weights α and β for determining the aggregated group similarity $g\_sim = \alpha \cdot avg\_sim + \beta \cdot e\_sim + (1 - \alpha - \beta) \cdot rel$ driving the selection of matching groups. Table 4 shows the results of the different weights. The quality of the group mapping highly depends on the edge similarity underlining the importance of considering the structural similarity within our household graphs. Without considering the edge similarity ($\beta = 0$), the F-measure for the group mapping drops to 90.7%, i.e. around 5.3% less than for the best configuration ($\alpha = 0.2, \beta = 0.7$) and also far less than when ignoring the record similarity ($\alpha = 0$). The uniqueness score can also improve the overall F-measure. For ($\alpha = 0.2, \beta = 0.7$) its weight is 0.1 which helped to achieve an improved F-measure compared to the three configurations where it is ignored (when the sum of α and β equals already 1). The best record mapping is also achieved for ($\alpha = 0.2, \beta = 0.7$) making it a good default configuration for our datasets.

5.2.3 Iterative vs non-iterative linkage

We now want to analyze to what degree the iterative group and record linkage with decreasing similarity thresholds is really helpful compared to a non-iterative, one-shot approach applying only a fixed minimal similarity threshold. To evaluate such a non-iterative approach we apply similarity functions with ω_2, $\delta\_high = 0.5$ and $\delta\_low = 0.5$ resulting in only one iteration. The results are shown in Table 5. We observe that the iterative approach indeed outperforms the non-iterative approach with an F-Measure improvement of $\approx 2.2\%$ for the group mapping and 3.1% for the record mapping. The improved quality mainly results from a substantially higher precision of more than 97% for both the group and record mapping. This is achieved because the iterative approach finds high-quality matches for the more restrictive thresholds while the more relaxed similarity threshold, with an increased risk of finding wrong matches, is limited to a subset of the records.

method		non-iterative	iterative
group mapping	Precision (%)	94.5	**97.3**
	Recall (%)	93.1	**94.8**
	F-measure (%)	93.8	**96.0**
record mapping	Precision (%)	91.8	**97.5**
	Recall (%)	93.1	**93.7**
	F-measure (%)	92.5	**95.6**

Table 5: Quality of the group mapping and record mapping by using the iterative vs. non-iterative approach.

5.3 Comparison with Existing Approaches

We compare our approach with two previously proposed methods: the collective entity resolution approach of [14] to determine a record mapping as well as the previous group linkage approach [8] for census data.

In [14], the authors propose a collective approach that is a specialization of [1]. It initially determines seed record links by applying a high record similarity. The seed links are used to incrementally identify additional links from the neighborhood of the linked records based on their attribute similarity and relational similarity. The overall algorithm follows a greedy strategy that selects in each iteration the record pair with the highest similarity. The related records update their similarities according to the selected record pair. In our implementation, we use the same similarity function as

method	CL	iter-sub
Precision (%)	93.5	**97.5**
Recall (%)	81.2	**93.7**
F-measure (%)	86.9	**95.6**

Table 6: Comparison of our approach with the collective linkage approach of [14] (CL) to determine a record mapping.

method	GraphSim	iter-sub
Precision (%)	**97.6**	97.3
Recall (%)	90.1	**94.8**
F-measure (%)	93.7	**96.0**

Table 7: Comparison of our approach with the household linkage approach of [8] (GraphSim).

in our approach (Table 2). Moreover, we filter all record pairs where the normalized age difference is more than 3 years[2]. To generate the seed link, we select the record links with a minimal similarity of 0.9. Table 6 shows the results of the record mapping obtained by collective linking. Our approach outperforms the collective approach w.r.t the record mapping quality by 8.6% for F-measure. The difference between our approach and the collective approach is that we can better link moved records with changed attribute values since we do not only link highly similar records (which is not sufficient for temporal linkage). Furthermore, our subgraph matching utilizes different relationships more comprehensively and benefits from incremental linkage.

The previous group linkage approach of [8] initially generates a highly selective record mapping consisting of 1:1 correspondences only. Based on this record mapping, the method calculates an average record similarity and an edge similarity between each group pair. Contrary to our approach, they calculate the similarities based on the initial 1:1 mapping. If correct record pairs are filtered out due to the 1:1 constraint, the approach is not able to identify these links. Hence, this filter step influences the average record similarity as well as the edge similarity, so that correct group links are not identified. Table 7 shows the results of the quality of the group mappings. Our approach achieves a significantly better F-measure for the group mapping compared to [8] (≈3.7%). This improvement is mainly because of a much higher recall that is limited in the previous approach mainly because of the use of the initial 1:1 mapping.

5.4 Analysis of Household Dynamics

Finally, we analyze the evolution of households from 1851 to 1901. For this purpose, we determine the evolution patterns for each successive census dataset pair based on the identified group and record mapping with the best parameter setting. Fig. 6 shows the frequency of each group evolution pattern for each pair of census datasets. In general, we observe an increasing number of households since the number of add_G patterns is higher than the number of $remove_G$ patterns for each new census. Moreover, we observe an increasing number of $preserve_G$ patterns due to the general increase in the number of households over time. From 1891 to 1901, there is also a high number of $remove_G$ patterns (up to ≈ 2200) indicating that many households may have moved to a new region. The complex patterns such as *split* and *merge* occur only rarely with an average occurrence of ≈ 100 for *split* and ≈ 70 while the *move* patterns are more frequent (≈ 1600 on average).

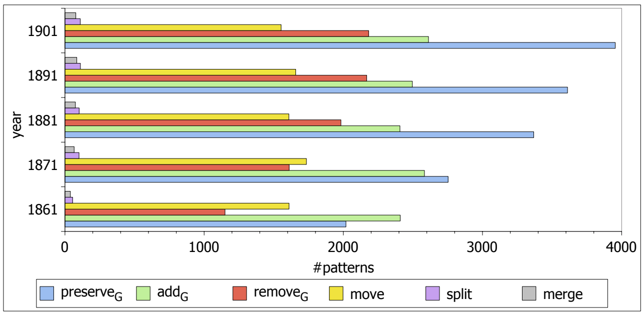


Figure 6: Quantitative Analysis of evolution patterns for census datasets from 1851 to 1901.

time interval	$\lvert preserve_G \rvert$
10	15705
20	7731
30	3322
40	1116
50	260

Table 8: Number of preserving households $\lvert preserve_G \rvert$ according to different time intervals (in years) from 1851 to 1901.

To analyze dependencies between households for the whole time period, we exploit the evolution graph and determine the largest connected component representing all households from 1851 to 1901 that are connected by group patterns. We identified the largest connected component with 17150 households over the complete interval from 1851 to 1901 thereby covering ≈52% of all households. Furthermore, we identify the number of preserved households according to different time intervals for the whole time period from 1851 to 1901. For instance, if we like to identify households that are preserved for 20 years, we define a graph pattern that consists of 2 edges with the pattern type $preserve_G$ since the difference between two census datasets is 10 years. Table 8 shows the number of preserved households for the different time intervals. The number of preserving households for all 10 year intervals (1851-61, 1861-71, 1871-81 etc.) represents the overall number of $preserve_G$ patterns of the quantitative analysis. Moreover, 260 household are preserved over the whole time period from 1851 to 1901.

6. RELATED WORK

Record linkage or entity resolution has been intensively studied in the past (see [4, 7, 12] for overviews). While the majority of approaches focus on evaluating the similarity of record attributes only, collective or context-based approaches additionally consider the similarity of relationships between entities for improved linkage decisions (e.g. [1, 8, 11, 14, 20, 23]). This idea has also been utilized in our approach but in a tailored way for use within groups such as households. Our approach is especially powerful as it considers different kinds of semantic relationships as well as the

[2] In our approach, subgraph matching ensures that such age differences are not accepted.

similarity of relationship attributes. Previous collective approaches have also not addressed temporal record linkage in contrast to our scheme.

Relatively few studies have investigated temporal record linkage (e.g., [2, 15, 17]) to link records within dynamically changing data. Existing approaches explicitly consider changing attribute values when matching individual records over time, e.g., by computing value transition probabilities [15]. Temporal clustering approaches as proposed in [3] group temporal records that belong to the same entity to reflect the entity history. Temporal record linkage approaches typically focus on matching individual person records while we also match groups of individuals and identify a record as well as group mapping to interconnect temporal records from census data.

Most closely related to our work is the group-based approach of [8] for matching households in historical census datasets. Our evaluation in Subsection 5.3 has shown that this previous scheme is outperformed by our approach due to its novel features such as an iterative group linkage and subgraph matching based on different semantic relationships. Richards and colleagues investigate in [21] the use of learning-based methods to optimize the use of attribute similarities for temporal record linkage (not group linkage) for census datasets. The observations of this study are complementary to ours and could be used for choosing alternate similarity functions for record matching.

Our work is further related to research on time and evolution-based analysis that is gaining increasing interest. For instance, there are studies analyzing historical web contents to find interesting patterns and trends [25], analyzing person histories on Twitter [16], or collecting and analyzing temporal knowledge from Wikipedia [24]. Our definition of change patterns is further related to previous work in the domain of ontology evolution [10, 22], in particular regarding change detection and diff computation (e.g. [9, 19]). These approaches typically identify basic and complex change operations between different ontology versions. We used this idea to identify time dependent patterns between groups of records to represent the semantics of changes in households over time. Based on the change patterns we are able to realize more comprehensive analysis, e.g., on complex evolution graphs.

7. CONCLUSIONS

We outlined and evaluated a new approach for temporal record and group linkage for the analysis of census data. The approach follows an iterative linkage strategy that first identifies high quality links thereby limiting the more error-prone identification of links between less similar records and groups to subsets of the input data. Group linkage is based on the identification of common subgraphs between groups such as households where we utilize the semantic relationships within groups and relationship properties such as the age differences between individuals. The evaluation showed the high effectiveness of the proposed approach that also outperforms a previous approach for linking census data.

We showed that the linkage results support a detailed evolution analysis of census data at both the level of individuals and groups. We proposed several evolution patterns to identify relevant changes including different kinds of group changes such as splits, merges and the movement of individuals from one group to another. All changes can be maintained within an evolution graph that can be used for a wide spectrum of change analysis, e.g., to identify frequent change patterns or to find connected groups over several census periods.

In future work, we plan to extend the change analysis of census data using the evolution graph and graph mining techniques. We also aim to apply and evaluate the proposed approach on larger census datasets. Furthermore, we want to study additional applications for group linkage, e.g., to analyze the changes in research teams or groups of coauthors over time.

8. ACKNOWLEDGEMENTS

This work was partially funded by the Australian Research Council (ARC) under Discovery Project DP160101934, and Universities Australia and the German Academic Exchange Service (DAAD).

References

[1] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):5, 2007.

[2] Y.-H. Chiang, A. Doan, and J. F. Naughton. Modeling entity evolution for temporal record matching. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1175–1186. ACM, 2014.

[3] Y.-H. Chiang, A. Doan, and J. F. Naughton. Tracking entities in the dynamic world: A fast algorithm for matching temporal records. *Proceedings of the VLDB Endowment*, 7(6):469–480, 2014.

[4] P. Christen. *Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection.* Springer Science & Business Media, 2012.

[5] P. Christen and R. W. Gayler. Adaptive temporal entity resolution on dynamic databases. In *Advances in Knowledge Discovery and Data Mining, 17th Pacific-Asia Conference PAKDD*, pages 558–569, 2013.

[6] X. L. Dong, A. Kementsietsidis, and W.-C. Tan. A time machine for information: Looking back to look forward. *ACM SIGMOD Record*, 45(2):23–32, 2016.

[7] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE TKDE*, 19(1):1–16, 2007.

[8] Z. Fu, P. Christen, and J. Zhou. A graph matching method for historical census household linkage. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 485–496. Springer, 2014.

[9] M. Hartung, A. Groß, and E. Rahm. Conto-diff: generation of complex evolution mappings for life science ontologies. *Journal of Biomedical Informatics*, 46:15–32, 2013.

[10] M. Hartung, J. F. Terwilliger, and E. Rahm. Recent advances in schema and ontology evolution. In *Schema Matching and Mapping*, pages 149–190. 2011.

[11] D. V. Kalashnikov and S. Mehrotra. Domain-independent data cleaning via analysis of entity-relationship graph. *ACM Transactions on Database Systems (TODS)*, 31(2):716–767, 2006.

[12] H. Köpcke and E. Rahm. Frameworks for entity matching: A comparison. *Data & Knowledge Engineering*, 69(2):197 – 210, 2010.

[13] H. C. Kum, A. Krishnamurthy, A. Machanavajjhala, and S. Ahalt. Social genome: Putting big data to work for population informatics. *Computer*, 47(1):56–63, 2014.

[14] S. Lacoste-Julien, K. Palla, A. Davies, G. Kasneci, T. Graepel, and Z. Ghahramani. Sigma: Simple greedy matching for aligning large knowledge bases. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 572–580. ACM, 2013.

[15] F. Li, M. L. Lee, W. Hsu, and W.-C. Tan. Linking temporal records for profiling entities. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 593–605, New York, NY, USA, 2015. ACM.

[16] J. Li and C. Cardie. Timeline generation: Tracking individuals on twitter. In *Proceedings of the 23rd international conference on World wide web*, pages 643–652. ACM, 2014.

[17] P. Li, X. Dong, A. Maurino, and D. Srivastava. Linking temporal records. *Proceedings of the VLDB Endowment*, 4(11):956–967, 2011.

[18] V. M. Moceri, W. A. Kukull, I. Emanual, G. van Belle, J. R. Starr, G. D. Schellenberg, W. C. McCormick, J. D. Bowen, L. Teri, and E. B. Larson. Using census data and birth certificates to reconstruct the early-life socioeconomic environment and the relation to the development of alzheimer's disease. *Epidemiology*, 12(4):383–389, 2001.

[19] N. F. Noy and M. A. Musen. PromptDiff: A fixed-point algorithm for comparing ontology versions. *AAAI/IAAI*, 2002:744–750, 2002.

[20] V. Rastogi, N. Dalvi, and M. Garofalakis. Large-scale collective entity matching. *Proceedings of the VLDB Endowment*, 4(4):208–218, 2011.

[21] L. Richards, L. Antonie, S. Areibi, G. W. Grewal, K. Inwood, and J. A. Ross. Comparing classifiers in historical census linkage. In *Proc. ICDM Workshops*, pages 1086–1094, 2014.

[22] L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. *User-Driven Ontology Evolution Management*, pages 285–300. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[23] A. Thor and E. Rahm. Moma-a mapping-based object matching system. In *CIDR*, pages 247–258, 2007.

[24] Y. Wang, M. Zhu, L. Qu, M. Spaniol, and G. Weikum. Timely yago: harvesting, querying, and visualizing temporal knowledge from wikipedia. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 697–700. ACM, 2010.

[25] G. Weikum, N. Ntarmos, M. Spaniol, P. Triantafillou, A. A. Benczúr, S. Kirkpatrick, P. Rigaux, and M. Williamson. Longitudinal analytics on web archive data: it's about time! In *CIDR*, pages 199–202, 2011.

In-DBMS Sampling-based Sub-trajectory Clustering

Nikos Pelekis
Dept. of Statistics and Ins. Science
University of Piraeus
Piraeus, Greece
npelekis@unipi.gr

Panagiotis Tampakis
Dept. of Informatics
University of Piraeus
Piraeus, Greece
ptampak@unipi.gr

Marios Vodas
Dept. of Informatics
University of Piraeus
Piraeus, Greece
mvodas@unipi.gr

Costas Panagiotakis
Dept. of Business Administration
TEI of Crete
Agios Nikolaos, Crete, Greece
cpanag@staff.teicrete.gr

Yannis Theodoridis
Dept. of Informatics
University of Piraeus
Piraeus, Greece
ytheod@unipi.gr

ABSTRACT

In this paper, we propose an efficient in-DBMS solution for the problem of sub-trajectory clustering and outlier detection in large moving object datasets. The method relies on a two-phase process: a voting-and-segmentation phase that segments trajectories according to a local density criterion and trajectory similarity criteria, followed by a sampling-and-clustering phase that selects the most representative sub-trajectories to be used as seeds for the clustering process. Our proposal, called S^2T-Clustering (for Sampling-based Sub-Trajectory Clustering) is novel since it is the first, to our knowledge, that addresses the pure spatiotemporal sub-trajectory clustering and outlier detection problem in a real-world setting (by 'pure' we mean that the entire spatiotemporal information of trajectories is taken into consideration). Moreover, our proposal can be efficiently registered as a database query operator in the context of extensible DBMS (namely, PostgreSQL in our current implementation). The effectiveness and the efficiency of the proposed algorithm are experimentally validated over synthetic and real-world trajectory datasets, demonstrating that S^2T-Clustering outperforms an off-the-shelf in-DBMS solution using PostGIS by several orders of magnitude.

CCS Concepts

• Information systems → Information systems applications → Data mining → Clustering • Information systems → Information systems applications → Spatio-temporal systems

Keywords

Mobility data mining; Sub-trajectory clustering; Trajectory segmentation; Trajectory sampling; MOD engines

1. INTRODUCTION

Knowledge discovery in mobility data [11][29][46][42] exposes patterns of moving objects exploitable in several fields. For instance, in both mature (transportation, climatology, zoology, etc.) and emerging domains (e.g. mobile social networks), scientists work with mobility-aware (mostly GPS-based) data, resulting in trajectories of moving objects stored in Moving Object Databases (MOD). Although during the recent years, there have been made significant achievements in the field [11][29][46][42], ongoing research calls for new methods aiming at deeper comprehension and analysis of mobility. For instance – and acting as motivation of this work – enhancing MOD engines, such as Secondo [1] and Hermes [31], with data mining operators is challenging [11][29] and is subject to the indexing extensibility interface of the corresponding ORDBMS on which they are implemented (see GiST [14][20], for example).

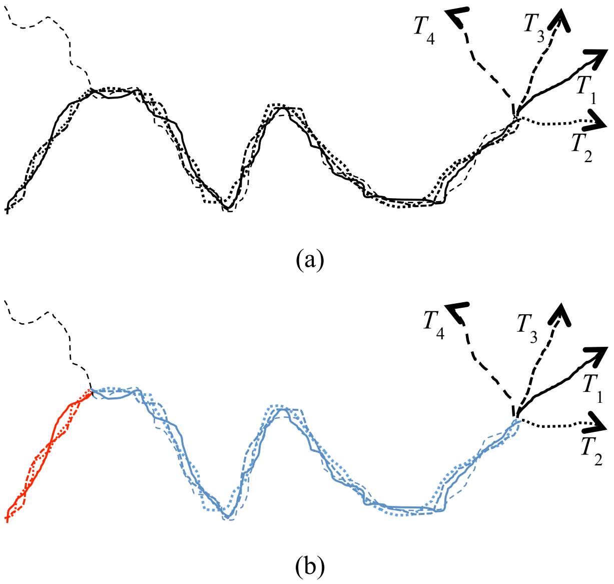
In the literature of trajectory-based mobility data mining, one can identify several types of mining models used to describe various collective behavioral patterns. As such, there exist works that identify various types of clusters of moving objects [10][26][21][32] and variations [4][17][22][44]. Related line of research is the one that builds representatives out of a trajectory dataset, either by generating artificial data [21][32] or by sampling the dataset itself [33][28].

Focusing on trajectory clustering, the majority of related work proposes a variety of distance functions, utilized by well-known clustering algorithms to identify collective behavior among whole trajectories [26][32][30]. A parallel line of research tries to discover local patterns in MOD, i.e. patterns that are alive only for a portion of moving objects' lifespan: some of those techniques simplify the given trajectories, however focusing on the spatial and ignoring the temporal dimension, such as TRACLUS [21], which is considered as the current state-of-the-art sub-trajectory clustering technique.

Figure 1 illustrates a working example that motivates our research: a dataset consisting of four trajectories, T_1, …, T_4. (In this figure, the time dimension is ignored for visualization reasons.) Among the sub-trajectories that compose the dataset, our goal is to identify two clusters (in red and blue, respectively) and five outliers (in black). In particular, the first (red) cluster consists of the tails of trajectories T_1, T_2 and T_3, the second (blue) cluster consists of the main bodies of trajectories of trajectories T_1, T_2, T_3 and T_4, while the rest portions of the trajectories (namely, the tail of T_4 and all four heads) are recognized as outliers.

Series ISSN: 2367-2005

10.5441/002/edbt.2017.84



(a)

(b)

Figure 1. (a) a MOD of 4 trajectories; (b) the MOD split in 2 clusters (in red and blue) and 5 outliers (in black).

Such clustering sounds impossible to be achieved by TRACLUS. This is due to the inherent design of that algorithm that, as delineated by the authors, discovers linear patterns only and fails to identify complex (e.g. snake-like) patterns like the ones that appear in Figure 1. In other words, when applied to this dataset, TRACLUS would eventually discover five to six linear clusters (one new cluster each time the snake-like motion changes direction). On the contrary, we wish to be able to follow these direction changes without assuming underlying constraints on the complexity of the shape of sub-trajectories found nor posing geometrical and temporal constraints, in terms of algorithm parameters, as those required by related work, e.g. [4][17]. For those having experimented with those techniques, parameters like disc radius, minimum duration and cardinality of patterns, are hard to be set in advance. For instance, a small detour of an object belonging to one of the clusters, would probably result in either the lack of those patterns or the formation of smaller ones.

Inspired by the above, in this paper we study an important problem in the mobility data management and exploration domain [29], that of sub-trajectory clustering and outlier detection. Informally, we aim at a methodology that builds clusters around (and detects outliers far away from) appropriately selected sub-trajectories that preserve the properties and the mobility patterns hidden in a MOD, as much as possible. Towards this goal, we introduce a novel clustering methodology exploiting on the voting, segmentation and sampling concepts proposed in [28]. More specifically, we devise an efficient voting process that allows us to describe the 'representativeness' of a trajectory in a MOD as a smooth continuous descriptor [28]. Using these descriptors (their 'representativeness'), we result in the automatic segmentation of trajectories into 'homogenous' sub-trajectories. Next, a deterministic sampling procedure selects only those sub-trajectories that optimally describe the entire MOD. Finally, we devise a method for sub-trajectory clustering driven by the aforementioned representative sample of sub-trajectories.

The design of such a clustering methodology is subject to two indispensible requirements that challenged our research: we seek for (a) an efficient and scalable solution that (b) should be able to operate on a real-world DBMS rather than being an ad hoc implementation using a sophisticated access method. This is in order for the proposal to be practical and useful in real-world application scenarios, where concurrency and recovery issues are taken into consideration. Both requirements call for a MOD engine; therefore, our proposal is implemented as a query operator in Hermes [16], implemented on top of PostgreSQL. To our knowledge, it is the first time in the literature that GiST is used to index trajectory-based mobility data for the above purposes. Therefore, we argue that this is an important step towards bridging the gap between MOD management and mobility data mining, as state-of-art approaches [25][40][12] could make use of the efficiency and the advantage of our proposal to execute in-DBMS clustering via simple SQL.

Our contribution is summarized below:

- we formulate the problem of sub-trajectory clustering (and outlier detection) in a MOD as an optimization problem;
- we propose an efficient solution, the so-called S^{2}T-Clustering algorithm, driven by a deterministic sampling methodology, with the number of clusters being automatically detected by the algorithm;
- in order to speed up clustering tasks in MOD systems, we implement S^{2}T-Clustering as a query operator over an expensible DBMS, namely PostgreSQL, based on access methods that exploit on the GiST indexing extensibility interface. (For validation purposes, we also implement S^{2}T-Clustering using PostGIS, an off-the-shelf in-DBMS alternative solution.)

The rest of the paper is organized as follows: Section 2 presents related work and Section 3 formulates the problem of sub-trajectory clustering (and outlier detection). Sections 4 and 5 present our proposal and its in-DBMS realization, respectively. Experimental results that evaluate S^{2}T-Clustering using synthetic and real trajectory datasets from urban and vessel traffic domains are provided in Section 6. Section 7 concludes the paper.

2. RELATED WORK

During the past decade, the field of MOD has emerged as a strong candidate for the efficient management of trajectory data exploiting on the robust architecture of extensible DBMS; Secondo [1] and Hermes [31] are typical examples of this paradigm. Nevertheless, extending a DBMS does not reduce the complexity of understanding their concurrency and recovery protocols, and as such, does not reduce the implementation effort of an external access method when compared to a built-in one, assuming that identical levels of concurrency, robustness and integration are desired [20]. Actually, complexity is the main reason that almost none of the numerous access methods for mobility data that have been proposed in the literature, [34][36][13] to name but a few representatives, have been integrated in a real Object-Relational DBMS. Even GiST [14] that has been proposed to serve access method extensibility has not been used so far in the context of mobility data. Mainly due to the above reasons, although a lot of research has been carried out in the field of MOD regarding efficient indexing and query processing, almost no related work exists in the field of mobility data mining in-DBMS [29].

Focusing on plain (i.e. outside DBMS) implementations, the common building block of trajectory clustering approaches is the use of different similarity functions as the means to group trajectories into clusters. Such a similarity function is proposed in [8] for the efficient processing of most-similar trajectory (MST) queries. T-OPTICS [26] incorporates a similar distance function into the well-known OPTICS [3]. In [5], probabilistic techniques based on EM algorithm are proposed for clustering (short) trajectories using regression mixture models. In [32], the authors propose CenTR-I-FCM, a variant of Fuzzy C-means (FCM) for MOD, while in [39] introduce the concept of uncertain group pattern. Both approaches propose specialized similarity functions having as goal to tackle the inherent uncertainty of trajectory data. In [8], the authors introduced the vector field k-means trajectory clustering technique whose central idea is to use vector fields to induce a notion of similarity between trajectories, letting the vector fields themselves define and represent each cluster. In [41], a multi-kernel-based estimation process leverages both multiple structural information within a trajectory and the local motion patterns across multiple trajectories in order to face challenges in case of large variations within a cluster and ambiguities across clusters. In [15], the Clustering and Aggregating Clues of Trajectories (CACT) pattern mining framework has been proposed for discovering trajectory routes that represent the frequent movement behaviors of a user. The approach exploits on a similarity measure for trajectories with silent durations (i.e., the time durations when no data points are available to describe the movements of users), which is used in a clue-aware clustering algorithm, where clues are some spatially and temporally close data points that capture certain common partial movement behaviors of the user.

TRACLUS [21] is a partition-and-group framework for clustering 2D trajectories (i.e. it ignores the time dimension), enabling the grouping of similar sub-trajectories, according to a trajectory partitioning step that uses the minimum description length principle. In its core, it uses a variant of DBSCAN [7], operating on the partitioned directed line segments. This work was the first to tackle the problem of identifying sub-patterns in trajectory data; however, it presents certain limitations (as discussed earlier) under the prism of the specifications we posed. In [24] the authors introduce an incremental trajectory clustering that exploits on TRACLUS.

Another line of research includes works that aim to discover several types of collective behavior among moving objects, forming a group of objects that moves together for a certain time period, such as moving clusters [18], flocks [4], convoys [17], swarms [23], traveling companion [36][37], gathering [44][45], and platoon [22] patterns. Although these approaches provide lucid definitions of the mined patterns, their main limitation is that they search for special collective behaviors, defined by respective parameters.

Our approach also finds commonalities to well-known approaches of clustering algorithms of point (vector) data [43][35], which sample the dataset at a pre-processing step and then perform the core clustering process aiming at high efficiency. However, these vector-based algorithms are not applicable to MOD due to the complex structure and properties of mobility data. Moreover, there is an essential difference between those techniques and our approach: while those mainly rely on random sampling, in our approach the clustering is driven by a sample resulted by an optimization formula, thus leading to a deterministic solution of the sub-trajectory clustering problem.

As already discussed, plain (sub-)trajectory clustering implementations leave concurrency and recovery outside the scene of requirements, as such setting limitations to their usage in real-world applications. In contrast, in this work we provide efficient in-DBMS solutions ready to be used by domain experts maintaining their volumes of data in state-of-the-art DBMS.

3. PROBLEM FORMULATION

Let $D = \{T_1, T_2, \ldots, T_N\}$ be a dataset consisting of N trajectories of moving objects (we assume that the objects move in the *xy*-plane). Let $p_{k,i} = (x_{k,i}, y_{k,i}, t_{k,i})$ be the *i*-th sampled point, $i \in \{1, 2, \ldots, L_k\}$ of trajectory T_k, $k \in \{1, 2, \ldots, N\}$, where L_k denotes the length of T_k (i.e. the number of points it consists of), the pair $(x_{k,i}, y_{k,i})$ and $t_{k,i}$ denote the 2D location and the time coordinate of point $p_{k,i}$, respectively. We consider linear interpolation between two successive sampled points, $p_{k,i}$ and $p_{k,i+1}$, so that each trajectory turns out to be a sequence of 3D line segments, $e_{k,i} = (p_{k,i}, p_{k,i+1})$, of cardinality $L_k - 1$, where each segment represents the continuous movement of the object during sampled points. Table 1 summarizes the definitions of the symbols used in this paper.

Table 1. Table of Symbols

Symbol	Definition
D	A dataset, $D = \{T_1, \ldots, T_N\}$, of N trajectories
T_k	k-th trajectory of D
$p_{k,i}$	i-th point of trajectory T_k, $p_{k,i} = (x_{k,i}, y_{k,i}, t_{k,i})$
L_k	Number of points forming trajectory T_k
$e_{k,i}$	i-th (3D) line segment of T_k, $e_{k,i} = (p_{k,i}, p_{k,i+1})$
LP_k	Number of sub-trajectories partitioning T_k
P_k	Set of the sub-trajectories partitioning T_k
$P_{k,i}$	i–th sub-trajectory of trajectory T_k
P	Set of sub-trajectories in dataset D, $P = \cup P_k$
V_k	Voting descriptor of trajectory T_k
V	Set of voting descriptors in dataset D, $V = \cup V_k$
$VP_{k,i}$	Voting descriptor of sub-trajectory $P_{k,i}$
$Nl_{k,i}$	Normalized lifespan descriptor of sub-trajectory $P_{k,i}$ w.r.t. lifespan of T_k
C	Clustering of sub-trajectories in M clusters, $C = \{C_1, \ldots, C_M\}$, $C_i \subset P$, $C_i \cap C_j = \varnothing$, $i \neq j$
S	Sampling set of representatives, $S = \{R_1, \ldots, R_M\}$, $S \subset P$, with sub-trajectory R_j representing cluster C_j
M	Cardinality of C (and S)
SR(S)	Representativeness function of S
$V(P_{k,i}, R_j)$	Voting descriptor of $P_{k,i} \in P{-}S$ w.r.t. sub-trajectory $R_j \in S$
Out	Set of outlier sub-trajectories, $Out = P{-}C$

Informally, the objective of sub-trajectory clustering is to partition trajectories into sub-trajectories and then form groups of similar ones, while at the same time, separating those that cannot fit in a group (called outliers). However, searching for entire trajectory similarity may be misleading since real-world trajectories may be long and consisting of heterogeneous portions of movement [6]. On the other hand, clustering at the sub-trajectory level sounds much more effective.

Rephrasing the previous discussion, if we consider trajectory T_k as a sequence of successive sub-trajectories $P_{k,i}$ of arbitrary length ($P_{k,i}$ is the *i*-th sub-trajectory of trajectory T_k), the objective of sub-trajectory clustering (and outlier detection) is to partition sub-trajectories into groups of similar ones and isolate the ones (called outliers) that are very dissimilar from the others. To achieve this, assuming a cluster is represented by its representative (or centroid) sub-trajectory, we define clustering as an optimization problem where the optimization criterion is to maximize the following expression:

$$SRD = \sum_{R_j \in S} \sum_{P_{k,i} \in C(R_j)} \overline{V(P_{k,i}, R_j)} \qquad (1)$$

The formula to be maximized, namely *Sum of Representativeness of Dataset* (SRD), uses set $S = \{R_1, \ldots, R_M\}$ of the representative sub-trajectories and the corresponding clusters $C(R_j)$ built around them, and is calculated upon $\overline{V(P_{k,i}, R_j)}$, i.e. the mean similarity (or average number of votes, according to our terminology) of sub-trajectory $P_{k,i}$ with respect to R_j.

Given the above formulation, the problem in hand is formalized as follows:

Problem 1 (Sub-Trajectory clustering in a MOD): *Assuming a dataset $D = \{T_1, T_2, \ldots, T_N\}$ consisting of N trajectories, where each of them is considered as a sequence P_k of successive sub-trajectories of arbitrary length, the problem of sub-trajectory clustering is defined as the task of partitioning the set $P = \cup P_k$ of sub-trajectories into (i) a clustering $C = \{C_1, \ldots, C_M\}$ of M clusters, $C_i \subset P$, $C_i \cap C_j = \varnothing$, $i \neq j$ (i.e. hard clustering), where each cluster is represented by its representative sub-trajectory $R_j \in P$, $j = 1, \ldots, M$, and (ii) a set Out of outliers, by maximizing Eq. (1).* ■

It is important to note that maximizing Eq. (1) is not trivial at all since one has to define, among others, (i) the criterion according to which a trajectory is segmented into sub-trajectories, (ii) the technique for selecting the set of the most representative sub-trajectories, (iii) whose cardinality *M* is unknown, to name but a few challenging sub-problems.

4. THE S^{2}T-CLUSTERING ALGORITHM

In this section, we propose a solution for Problem 1 defined above, which is called S^{2}T-Clustering (for Sampling-based Sub-Trajectory Clustering). Our proposal (listed in Algorithm 1) consists of two phases: first, we apply the so-called *Neighborhood-aware Trajectory Segmentation* (aka NaTS) method that is able to detect homogenized sub-trajectories applying trajectory voting and segmentation; then, we apply the so-called *Sampling, Clustering, and Outlier detection* (aka SaCO) method that selects the most representative among the sub-trajectories detected in the previous phase in order for them to serve as the seeds of the clusters to be produced.

Algorithm 1. S^{2}T-Clustering

```
Input: trajectory dataset $D = \{T_1, T_2, \ldots, T_N\}$, voting influence
$\sigma$, threshold $\varepsilon$
Output: sampling set $S$, clustering $C$, set of outliers $Out$.
    // Initialization phase
1.  Reset set $V$ of voting descriptors in $D$
    // NaTS phase (Neighborhood-aware Trajectory
    Segmentation)
2.  for each trajectory $T_k \in D$ do
3.      Update set $V$ of voting descriptors in $D$ w.r.t. $T_k$ and $\sigma$
4.      Partition $T_k$ in set $P_k$ of sub-trajectories w.r.t. $V_k$
    // SaCO phase (Sampling, Clustering, and Outlier
    detection)
5.  Find sampling set $S$ consisting of the $M$ most
    representative sub-trajectories
6.  Using set $S$ and threshold $\varepsilon$, partition $P = \cup P_k$ in a set $C$ of
    $M$ clusters and a set $Out$ of outliers
7.  return ($S$, $C$, $Out$)
```

It is important to note that the number *M* of representatives (hence, the number of clusters) is not user-defined; rather, it is the algorithm that estimates it (in Line 6). As for parameters σ and ε that appear in Algorithm 1 (Line 3 and Line 7, respectively), *σ* controls how fast the voting influence decreases with distance, whereas ε acts as a lower bound threshold of similarity between representative and non-representative sub-trajectories, thus deciding whether a (non-representative) sub-trajectory will be flagged as outlier or not. These parameters will be explained in detail in the subsections that follow.

4.1 NaTS: Neighborhood-aware Trajectory Segmentation

We extend the concept of density-biased sampling (DBS), which was originally proposed for point datasets [18], to be applied to trajectory segments. According to DBS, the local density for each point of a set is approximated by the number of points in a surrounding region, divided by the volume of the region. In our case, adopting a voting process of trajectories in MOD as defined in [28], we define the representativeness of a 3D trajectory segment $e_{k,i}$ of a given trajectory T_k to be the number of 'votes' this segment collects from other trajectories w.r.t. their mutual distance. The overall voting collected by a segment (a value ranging from 0 to *N*) has the physical meaning of the number of other trajectories that co-exist with the trajectory that segment belongs to, both spatially and temporally. Intuitively, the voting results can be post-processed in order for us to be able to identify homogeneous (w.r.t. representativeness) sub-trajectories.

Formally, let V_k be the voting trajectory descriptor along the line segments of T_k, consisting of a series of L_k–1 components. Each component $V_{k,i}$ of this vector corresponds to the number of votes ("representativeness" value) that segment $e_{k,i}$, $i \in \{1, \ldots, L_k–1\}$, collected by the segments of the other trajectories. This representativeness value is based on a distance function $d(e_{k,i}, e_j)$ between two line segments $e_{k,i}$ and e_j, $k \neq j$. This distance function is defined as the definite integral of the time-varying distance $D_j(t)$ between the two segments during their common lifespan $[t_{j,start}, t_{j,end})$, following the approach proposed in [8]:

$$d(e_{k,i}, e_j) = \int_{t_{j,start}}^{t_{j,end}} D_j(t)\, dt \qquad (2)$$

As D_j follows a trinomial, this integral is efficiently approximated by the Trapezoid Rule:

$$\left(D_j(t_{j,start}) + D_j(t_{j,end})\right) \cdot (t_{j,start} - t_{j,end})/2$$

and can be computed in *O*(1), as it has been already proved in [8].

Given the above distance function, the representativeness value is provided by the following voting function.

$$V(e_{k,i}, e_j) = e^{-\frac{d^2(e_{k,i}, e_j)}{2\cdot\sigma^2}} \qquad (3)$$

As already mentioned, parameter $\sigma > 0$ controls the "voting influence", i.e. how fast $V(e_{k,i}, e_j)$ decreases with distance. It also holds that $V(e_{k,i}, e_j)$ is bounded in [0, 1]: it gets value 1 when the distance of the two segments is zero (i.e. the segments are identical) while very high distance results in voting value close to zero.

After the voting process takes place, the trajectory segmentation process gets into action. The goal of this step is to partition each trajectory into *homogenous representativeness* sub-trajectories, irrespectively of their shape complexity (recall the discussion about the snake-like trajectories in Figure 1). In order to perform neighbourhood-aware trajectory segmentation, we adopt the *Trajectory Segmentation Algorithm* (TSA), proposed in [28]. In other words, the result of the voting process is given as input to TSA, which provides as output the sub-trajectories along with their voting descriptors. More technically, let $P_{k,i}$, $i \in \{1, \ldots, LP_k\}$, be the *i*-th sub-trajectory of T_k, where LP_k denotes the number of partitions of T_k. Then, $VP_{k,i}$ is the voting descriptor formed by the representativeness values of the segments that belong to $P_{k,i}$. In other words, $VP_{k,i}$ shows how many trajectories find themselves to be similar to $P_{k,i}$. The interested reader is referred to [28] for the technical details of TSA.

Back to the example of Figure 1, the NaTS phase results in segmenting trajectory T_1 into three sub-trajectories (coloured red, blue, and black, respectively, in Figure 1(b)); similar for the other trajectories of the dataset. Thus, the overall result of this phase consists of 12 sub-trajectories along with their voting descriptors.

4.2 SaCO: Sampling, Clustering, and Outlier detection

As already mentioned, trajectory segmentation aims to provide homogeneous sub-trajectories according to their representativeness, i.e. with respect to their local similarity with other trajectories. On the other hand, the goal of sub-trajectory clustering is to partition the dataset into groups (clusters) of similar sub-trajectories. Therefore, in our proposal, we first select the appropriate sampling set S and then tackle the problem of clustering according to the following idea (quite popular, also in traditional data clustering): *each sub-trajectory in the sampling set is considered to be a representative around which a cluster will be formed*. So, our goal is that the sampling set should contain highly voted trajectories of the MOD which, at the same time, would cover the 3D space occupied by the entire dataset as much as possible in order for Eq. (1) to be maximized.

In order to achieve this goal, we propose the sampling to be done by maximizing a formula (see Eq. (4)) that would take into account the votes $VP_{\mathrm{k,i}}$ collected by each sub-trajectory. Formally, let S denote the sampling set, so that $S_{\mathrm{k,i}}$ is one, if sub-trajectory $P_{\mathrm{k,i}}$ belongs to the sampling set, and zero otherwise. According to the previous discussion, the number of sub-trajectories that are represented in the sampling set S, should be maximized. This is formalized in Eqs. (4)-(6).

$$SR(S) = \sum_{k=1}^{N} \sum_{i=1}^{LP_k} S_{k,i} \cdot SR_{gain}(k,i) \tag{4}$$

where

$$SR_{gain}(k,i) = \sum_{j=1}^{|P_{k,i}|} VP_{k,i,j}^{P} \cdot Nl_{k,i,j} \cdot (1 - VP_{k,i,j}^{S}) \tag{5}$$

$$Nl_{k,i,j} = lifespan(e_{k,i,j})/lifespan(T_k) \tag{6}$$

More precisely, $SR_{gain}(k,i)$ expresses the gain in $SR(S)$ if we add $P_{\mathrm{k,i}}$ in S, $|P_{\mathrm{k,i}}|$ denotes the number of line segments of $P_{\mathrm{k,i}}$, $VP_{k,i,j}^{P}$ and $VP_{k,i,j}^{S}$ denote the votes in P and the votes in S, respectively, of the j-th line segment of $P_{\mathrm{k,i}}$ and are calculated according to Eq. (3). As for $Nl_{\mathrm{k,i}}$, it denotes the normalized lifespan descriptor of sub-trajectory $P_{k,i}$ w.r.t. lifespan of T_{k}, namely $Nl_{\mathrm{k,i,j}}$ is the fraction of the duration of the j-th line segment of $P_{\mathrm{k,i}}$ with respect to whole lifespan of T_{k}.

For this purpose, we follow the ideas included in the *Sub-trajectory Sampling Algorithm* (SSA), proposed in [28]. However, SSA is not appropriate for an efficient in-DBMS solution, which is one of our main objectives. Thus, we keep the main characteristics of the algorithm and adapt it in order to meet our specifications (described in detail in Section 5.2). In principle, the input of sampling algorithm is the set P of all sub-trajectories P_{k}, the set voting $VP_{\mathrm{k,i}}$ and the normalized lifespan $Nl_{\mathrm{k,i}}$ vectors of these sub-trajectories, all provided by the NaTS phase. The output of the sampling step is the sub-trajectory sampling set S consisting of M samples. Back to the example of Figure 1, this step results in selecting two sub-trajectories (samples), one out of the three red and one out of the four blue sub-trajectories.

As already mentioned, the population M of the samples is not user-defined; in contrary, it is dynamically estimated by SSA algorithm. As such, it provides a deterministic solution, in contrast to other probabilistic [18][27] or user-supervised, explorative sampling techniques [2].

What follows is the clustering step, which takes into account the sampling set S and the vector of votes (i.e. representativeness) $V(P_{\mathrm{k,i}}, R_{\mathrm{j}})$ between, on the one hand, the non-representative $P_{\mathrm{k,i}} \in P{-}S$ and, on the other hand, the representative sub-trajectories $R_{\mathrm{j}} \in S$. Technically, $V(P_{\mathrm{k,i}}, R_{\mathrm{j}})$ consists of $|P_{\mathrm{k,i}}|$ elements, where each element represents the voting that takes place between the segments of $P_{\mathrm{k,i}}$ and R_{j}. As illustrated in Eq. (1), we use the mean value $\overline{V(P_{k,i},R_j)}$ of the vector values $V(P_{\mathrm{k,i}}, R_{\mathrm{j}})$. Each of those values is computed by measuring the distance of the corresponding segment of $P_{\mathrm{k,i}}$ from its nearest to R_{j} and then by applying the voting function of Eq. (3). Thus, it holds that $0 \le \overline{V(P_{k,i},R_j)} \le 1$.

Concluding the discussion about Algorithm 1, in order to find the clusters that maximize Eq. (1), the sub-trajectories that are assigned to cluster $C(R_{\mathrm{j}})$ represented by sub-trajectory $R_{\mathrm{j}} \in S$, are the ones that fulfil the following property:

$$C(R_j) = \left\{P_{k,i} \in P - S\colon \overline{V(P_{k,i},R_j)} \ge \overline{V(P_{k,i},R_v)}\ \forall R_v \in S \wedge \overline{V(P_{k,i},R_j)} \ge \varepsilon\right\} \tag{7}$$

and

$$C = \cup\, C(R_j) \tag{8}$$

On the other hand, the sub-trajectories that are considered outliers (thus forming the outliers set *Out*) are those failing to be assigned to a cluster, formally:

$$Out = P - C \tag{9}$$

As already discussed, parameter ε controls how far from a representative a non-representative should be positioned in order for the latter to be flagged as outlier. Back to the example of Figure 1, the clustering process presented above results in two clusters, formed around the red and the blue, respectively, representative sub-trajectory found in the sampling step. As a side effect, the black sub-trajectories are left out of the two clusters, thus they are flagged as outliers.

5. S^{2}T-CLUSTERING IN-DBMS

In this section, we present our methodology for the efficient in-DBMS development of S^{2}T-Clustering algorithm proposed in Section 4.

5.1 NaTS in-DBMS

NaTS phase of S^{2}T-Clustering algorithm (Lines 2–4 in Algorithm 1) consists of two steps: (a) voting among trajectory segments and (b) trajectory segmentation based on the resulted voting descriptors. An efficient in-DBMS solution should focus on the voting step (Lines 2–3), since TSA [28] that implements the segmentation step (Line 4) poses no special challenges; it is an efficient in-memory algorithm applied only on the voting descriptor of a single trajectory.

Back to the voting step, to meet its requirement we need an algorithm that takes as input a dataset $D = \{T_1, T_2, \ldots, T_N\}$ of trajectories, a trajectory $T_{\mathrm{k}} \in D$ and $\sigma > 0$ parameter, and provides as output a voting descriptor (vector) V_{k} consisting of $L_{\mathrm{k}}{-}1$ components, each corresponding to segment $e_{\mathrm{k,i}}$, $i \in \{1, \ldots, L_{\mathrm{k}}{-}1\}$, of trajectory T_{k}. For efficiency purposes, [28] implemented the demanding voting process by using an incremental nearest neighbour (INN) algorithm. However, given the specifications posed in the introduction of this paper, INN is not a choice due to the fact that the access methods supported by real ORDBMS (e.g. the GiST interface in PostgreSQL) do not support the incremental paradigm. This implies that, in our case, we are directed to queries natively supported by ORDBMS, such as typical range and NN queries.

Let us now discuss the design and implementation options we have in-DBMS. Dataset D corresponds to a relation with tuples in the form <*t_id*, *s_id*, $e_{\mathrm{k,i}}$>, where *t_id* (*s_id*) is the trajectory (segment, respectively) identifier and $e_{\mathrm{k,i}}$ corresponds to the 3D segment, upon which a 3D-R-tree index is built. Nevertheless, this setting is straight-forwardly realized in the well-known PostGIS spatial extension of PostgreSQL using 3D GiST. (Note, however, that PostGIS handles time- dimension as simply as a (third) z- spatial dimension, next to x- and y- dimensions.) An important issue has also to do with the realization of Eq. (3) that provides the voting between two segments: theoretically, a segment may vote (though close to zero) even if it is found very far from the target segment. However, this is not realistic in DBMS implementations. As such, we introduce *s_buffer*, a spatial threshold for distance between two segments, above which there is no need to calculate this distance. In the case where the application user has limited knowledge about space-time properties of the dataset, this parameter can be tuned to be the maximum value resulting in a very low (close to zero) voting as computed by Eq. (3). This is achieved as follows: by reversing Eq. (3), we obtain Eq. (10) that defines an upper bound for *s_buffer*.

$$d \leq \sqrt{-2\sigma^2 \cdot \ln(\varepsilon)} \qquad (10)$$

Thus, d values higher than the upper bound set in Eq. (10) are not expected to contribute to the quality of the clustering.

Given the above setting, voting can be implemented using at least two alternatives, called Baseline-I and Baseline-II, respectively. Baseline-I solution performs $\sum_k (L_k - 1)$ range queries in the 3D-R-tree, where each query window corresponds to the MBB of a segment, enlarged by *s_buffer*; hence, the total number of range queries equals to the total number of segments in D, a fact that turns this solution to be expensive in disk accesses. On the other hand, Baseline-II solution performs N range queries in the 3D-R-tree, where each query window corresponds to the MBB of a trajectory, again enlarged by *s_buffer*; hence, the total number of range queries equals to the number of trajectories in D. Obviously, the second solution is much cheaper in disk accesses regarding the index but, unfortunately, imposes a heavy refinement step because of the volume of the trajectory MBB. Anyway, both approaches need a refinement step to calculate voting descriptor $V_{k,i}$, which involves distance calculations.

In the following paragraphs, we present an alternative (third) approach for addressing the voting step, which is the most demanding step in S^{2}T-Clustering algorithm and, as such, it needs special care. In particular, we follow a filter-and-refinement approach that utilizes a range-like query, called *Trajectory Buffer Query* (TBQ). TBQ takes as input a trajectory, enlarges it by *s_buffer*, and returns the segments that overlap with the sequence of the enlarged MBBs of the trajectory's segments. The TBQ rationale is to efficiently retrieve those segments in D that are "around" a given trajectory, where "around" is defined by *s_buffer*. Figure 2 illustrates the Trajectory Buffer TB_k of a trajectory T_k.

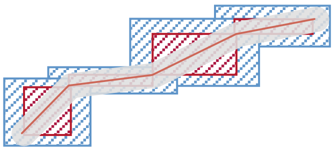


Figure 2. The Trajectory Buffer TB_k (i.e. the sequence of the blue MBBs) of a trajectory T_k.

It is obvious that our proposal follows a trajectory-based approach (i.e. similar to the Baseline-II technique), but for each trajectory it minimizes the filtering step by diminishing the dead space of the query, and thus minimizes the expensive refinement step. In turn, this implies changing the default search strategy of the 3D-R-tree over GiST that will reduce the time needed to compare a node entry with the trajectory buffer that is passed as predicate to the index. This is achieved by the *Consistent* method of the GiST extensibility interface [14], which contains the comparison logic between an index node entry of GiST and the trajectory buffer. Algorithm 2 outlines TBQ whereas Algorithm 3 presents the adapted *Consistent* method of the GiST interface.

Algorithm 2. Trajectory Buffer Query (TBQ)

```
Input: pg3D-R-tree root, trajectory $T_k$, parameter s_buffer
Output: set of segments that overlap with $TB_k$
    $TB_k$ ← TrajectoryBuffer($T_k$, s_buffer)
    root.depth-first-search(Consistent, $TB_k$)
```

Algorithm 3. Consistent

```
Input: Trajectory Buffer $TB_k$, current index entry E
Output: Boolean
1.  if E is in a leaf node then
2.      if MBB(E.segment) overlaps MBB($TB_k$) then
3.          for each $MBB_i \in TB_k$ do
4.              if E.segment overlaps $MBB_i$ then
5.                  return true
6.  else // E is in a non-leaf node
7.      if E.box overlaps MBB($TB_k$) then
8.          for each $MBB_i \in TB_k$ do
9.              if E.box overlaps $MBB_i$ then
10.                 return true
11. return false
```

Recall that *Consistent* decides whether the depth-first search should visit a child of the current entry or not (if the entry belongs to a non-leaf node) or, in case the entry belongs to a leaf node, checks whether to return the segment pointed by the leaf entry. After this remark, the depth-first search driven by *Consistent* in Algorithm 3 is easy to be followed: *Consistent* returns true if the MBB of the entry overlaps with one of the MBBs forming the trajectory buffer TB_k (Lines 5 and 10, for leaf and non-leaf nodes, respectively). Before this check takes place, a brute filtering is applied by checking whether the MBB of the entry overlaps the entire MBB of TB_k (Lines 2 and 7, respectively).

5.2 SaCO in-DBMS

In this section, we discuss the in-DBMS development of SaCO, i.e. the second phase of S^{2}T-Clustering. SaCO phase (Lines 5–6 in Algorithm 1) also consists of two steps: (a) sampling of the most representative sub-trajectories (Line 5) and (b) clustering around samples and outlier detection (Line 6).

Regarding the sampling step, we adopt the SSA algorithm [28] as a starting point and we improve it with two crucial modifications, focusing on the efficiency and the quality, respectively, of the samples selected. The first improvement is that the voting method that is inherent in the sampling process follows the much more efficient approach presented earlier rather than the one presented in [28]. The second modification is about the selection of an even better set of representatives; as proposed in [28], SSA selects representatives as long as (a) the top-k number of representatives is less than a user-defined threshold (i.e. parameter M that acts as an upper bound for the selected representatives) and (b) the optimization criterion is satisfied (see Eq. (4) and (5)). In fact, SSA selects the highly voted sub-trajectories, while at the same time it tries to penalize sub-trajectories that are very close to already selected representatives. Sometimes this automatic penalization fails, resulting to very similar representatives. In contrast, in our case, as the representatives are employed as cluster pivots, when a new representative is selected, it is further examined whether it is similar with one of the already selected representatives. In such a case, it is not selected and the algorithm evaluates the next candidate sub-trajectory. The similarity criterion is the same with the one adopted for the clustering, i.e. Eq. (7).

What follows is the final step, that of clustering and outlier detection. For this purpose, we follow an index-based, greedy approach that takes advantage of the TBQ query, which is applied on the results of the SSA algorithm, so as to form clusters around the sampled sub-trajectories. To this end, we propose the so-called *Sub-trajectory Clustering Algorithm* (SCA). SCA, listed in Algorithm 4, receives as input set *P* of sub-trajectories, set *S* of representatives, as it was produced by the (modified) SSA, and threshold parameter ε. The output of the method is the final result of S^2T-Clustering, i.e. sets *C* and *Out*, with the clusters and outliers, respectively.

Algorithm 4. SCA

```
Input: set $P$ of sub-trajectories, set $S$ of representatives, parameter $\varepsilon$
Output: set $C$ of clusters, set $Out$ of outliers
1.  $Out = P - S$
2.  for each $R_j \in S$ do
3.      $C_j \leftarrow \{R_j\}$
4.  for each $R_j \in S$ do
5.      $TBQ_j \leftarrow TBQ(Out, R_j, s\_buffer)$
6.      for each $e_{j,f} \in R_j$ do
7.          $TBQ_{j,f} \leftarrow$ overlaps($TBQ_j$, extend($e_{j,f}$, $s\_buffer$))
8.      for each $P_{k,i}$ in $\{TBQ_{j,f}\}$, $f \in [1, |R_j|]$ do
9.          $v \leftarrow \overline{V(P_{k,i}, R_j)}$
10.         if $v > \varepsilon$ and $v > old\_v_{k,i}$ then
11.             $C_j \leftarrow C_j \cup \{P_{k,i}\}$
12.             flag $P_{k,i}$ as clustered in $Out$
13.             $old\_v_{k,i} \leftarrow v$
14. for each $P_{k,i}$ in $Out$ do
15.     if $P_{k,i}$ is flagged as clustered then
16.         $Out \leftarrow Out - \{P_{k,i}\}$;
17. return ($C$, $Out$)
```

Initially, the sub-trajectories are organized in two sets (implemented as relations in DBMS), one containing the sampling set sorted by the order of their selection and the other containing the remaining data, while each cluster is initialized by a representative sub-trajectory from the sampling set. As such, each representative sub-trajectory constitutes the first member (seed) of the corresponding cluster (Lines 1-3). Then, we apply a two-step filtering procedure so as to increase the efficiency of the algorithm. At the first step, for each cluster seed $R_{j,}$ we apply a TBQ query, which returns the segments that are "close" to the cluster seed (Line 5). Subsequently, for each segment $e_{j,f}$ belonging to the specific representative R_j, we apply a spatiotemporal range query with the same spatial component as that of the TBQ query (Line 7). This spatiotemporal range query is performed in order to identify the segments that are "close enough" to $e_{j,f}$ and, hence, qualify to proceed to the voting procedure w.r.t. R_j. Subsequently, for each non-clustered $P_{k,i}$, we calculate the average voting that R_j receives (Line 9). By taking into account parameter *ε* discussed earlier, we assign it to cluster C_j mastered by R_j (Line 11) and mark it as clustered (Line 12). Through this process, in the case where $P_{k,i}$ belongs to the result of more than one TBQ searches, it is assigned to the representative that has achieved the highest voting.

6. EXPERIMENTAL STUDY

In this section, we present the results of our experimental study. All experiments were conducted on an Intel Xeon X5675 Processor 3.06GHz with 48GB memory, running on Debian Release 7.0 (wheezy) 64-bit. The proposed algorithms were implemented on top of a PostgreSQL 9.4 server with the default configuration for its memory parameters. We should clarify that in our implementation, which exploits on the extensibility interface given by PostgreSQL, we have defined and implemented from scratch datatypes and operands conforming to the whole discussion so far, resulting in the so-called Hermes@PostgreSQL [16], which is completely independent from PostGIS. This implies that the 3D-R-tree has also been implemented from scratch (on top of GiST); we call it pg3D-R-tree (see the input of TBQ in Algorithm 2).

A notable difference of our pg3D-R-tree from the PostGIS implementation of the 3D-R-tree is that, in our case, the entries of the leaf nodes are 3D segments rather than 3D boxes. This is an implicit assumption in the *Consistent* algorithm (see e.g. Line 2 in Algorithm 3), which allows us to avoid additional I/O operations. The outline of our experimental study is as follows: First, we study the robustness of S^2T-Clustering by using a synthetic dataset (where we know the ground truth) in order to (a) evaluate the sensitivity of our proposal w.r.t. various parameters and (b) validate whether our approach succeeds to discover the underlying clusters (and outliers). Then, a set of experiments is performed in order to evaluate the efficiency and scalability of S^2T-Clustering. These experiments are performed using three different approaches: the two baseline solutions and our solution based on TBQ, as they were presented in Section 5.

6.1 Datasets

The three datasets we used in our experimental study, one synthetic (SMOD) and two real datasets (IMIS, GeoLife), are presented in the following paragraphs.

SMOD - Synthetic MOD (SMOD)[1] consists of 400 trajectories and is used for the ground truth verification (see the discussion about ground truth below). The creation scenario of the synthetic dataset is the following: the objects move upon a simple graph that consists of the following destination nodes (points) with coordinates: A(0,0), B(1,0), C(4,0) and D(2,1). Half of the objects move with normal speed (2 units per second) and another half move with high speed (5 units per second). Figure 3 illustrates the 2D map of the SMOD consisting of three one-directional (A → B, B → D, D → C) and one bi-directional road (B ⇆ C). All objects move under the following scenario, for a lifetime of 100 seconds:

- (normal movement – 99% of the trajectories) All objects start from point A towards point B; the high-speed objects start at t = 0 sec and the normal-speed objects start at t = 20 sec. When an object arrives at B, it ends its trajectory with a probability of 15%; otherwise, it continues with the same speed to the next point. If there exist more than one option for the next point, it decides randomly about the next destination.
- (abnormal movement – 1% of the trajectories) A few outlier objects follow a random movement in space (other than these roads) with a speed that is updated randomly.

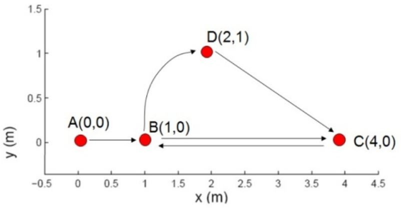


Figure 3. The 2-D map of SMOD.

The ground truth of the clusters that are hidden in SMOD can be inferred by the description of the dataset itself. In particular, eight clusters of sub-trajectories (as well as a set of outliers) are identified. Table 2 lists the eight clusters along with their spatial (2nd column) and temporal projection (3rd column).

[1] Publicly available at chorochronos.datastories.org repository under the name 'smod'.

Table 2. The ground truth hidden in SMOD

Cluster	Path	Time periods (clusters)
#1, #2	A→B	[0, 0.2], [0.2, 0.7]
#3, #4	B→C	[0.2, 0.8], [0.7, 1.2]
#5, #6	B→D	[0.2, 0.52], [0.7, 1.2]
#7	C→B	[0.8, 1]
#8	D→C	[0.52, 1]

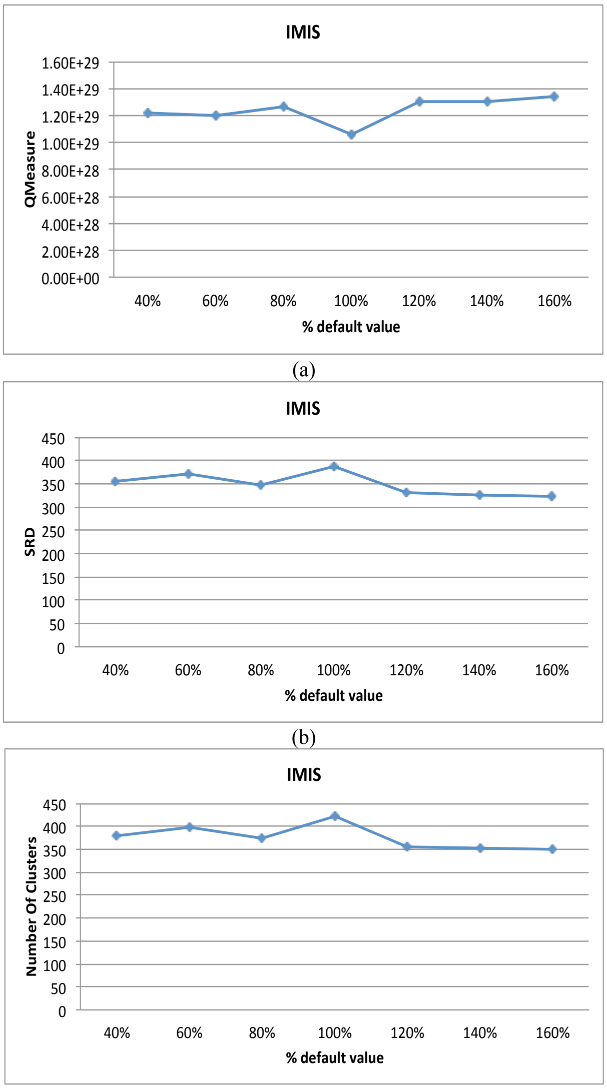
As for real datasets, **GeoLife** [47] consists of the trajectories of 178 users in a period of more than four years; this dataset represents a wide range of movements, including not only urban transportation (e.g. from home to work and back) but also different kinds of activities, such as sports activities, shopping, etc. Finally, **IMIS**[2] is a real AIS dataset consisting of the trajectories of 637 ships moving in the Greek seas for one week. Table 3 presents the statistics of the three datasets.

6.2 Quality of Clustering Analysis

In this section, we perform a sensitivity analysis in order to explore the effect on the quality of clustering when setting different values on certain parameters. The quality of the clustering is calculated through two different measures: QMeasure [21] and SRD (see Eq. (1)). We should mention that the lower the QMeasure the higher the quality; on the other hand, the higher the SRD the higher the quality. Regarding parameter settings, as our approach shares similar concepts with the sampling methodology of [28], we followed the best practices presented in that work. More specifically, parameter σ was set to 0.1% of the dataset diameter while ε was set to 10^{-3}. Regarding *s_buffer*, it was automatically set according to Eq. (10) as default value and we experimented with values around the default.

Table 3. Dataset Statistics

Statistic	SMOD	GeoLife	IMIS
# Trajectories	400	18,668	5110
# Segments	35,273	24,159,325	443,657
Dataset Duration (hh:mm:ss)	0:02:00	1932 days 22:59:48	6 days 19:59:53
Avg. Sampling Rate (hh:mm:ss)	0:00:01	0:00:08	0:18:02
Avg. Segment Length (m)	8	72	1545
Avg. Segment Speed (m/s)	7.83	5.01	7.03
Avg. Trajectory Speed (m/s)	2.86	3.91	4.52
Avg. # Points per Trajectory	89	1295	88
Avg. Trajectory Duration (hh:mm:ss)	0:01:28	2:43:15	11:33:45
Avg. Trajectory Length (m)	691	93,046	134,148



(a)

(b)

(c)

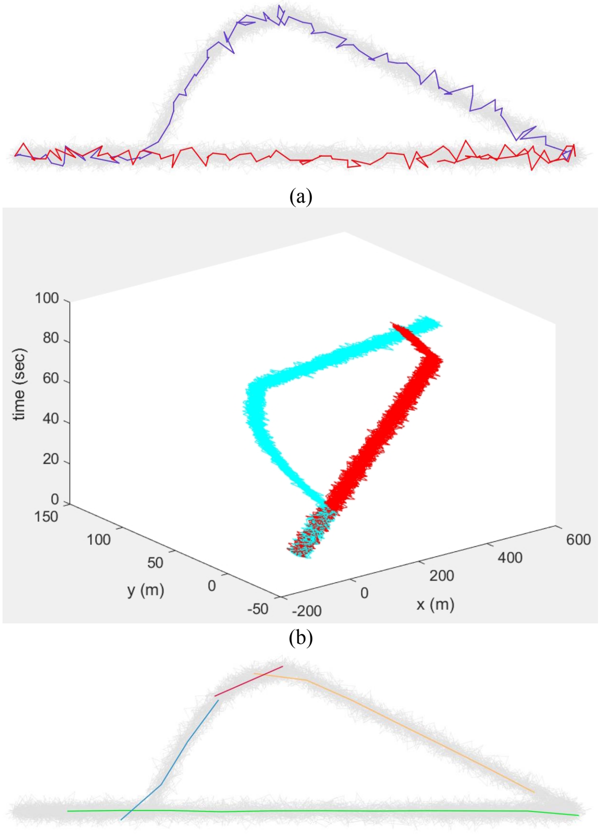
Figure 4. The effect on (a) QMeasure, (b) SRD, (c) the discovered number of clusters, when varying *s_buffer* parameter around its default value.

The first set of experiments is about the sensitivity of S^2T-Clustering w.r.t. *s_buffer*. Figure 4 illustrates the results over the IMIS dataset. In particular, we used the default value (labelled 100% in the x-axis of the charts) as well as 6 values around it (labelled 40%, 60%, 80%, 120%, 140%, 160%). As one can easily observe, the quality of the clustering, measured either by QMeasure or SRD, remains more or less stable and follows the trend of the number of clusters identified. Moreover, in both QMeasure and SRD, the best quality appears when *s_buffer* is set to its default value (*d*).

We repeated the same experiment over GeoLife and resulted in similar conclusions. Considering the above analysis, the value for *s_buffer* used in the remainder of our experimental study is the default value provided by Eq. (10).

In a second set of experiments, we applied our proposal to the SMOD dataset, which is ideal for the purposes of testing the quality of our algorithm. In order to measure the stability of our method to noise effects, we have added Gaussian white noise of different Signal to Noise Ratio (SNR) levels, measured in db, to the spatial coordinates of SMOD. All the subsequent experiments have been repeated with SNR = 30db and SNR = 50db and the results were the same. Therefore, we present only the case with the SNR =30db.

[2] Publicly available at chorochronos.datastories.org under the name ‘imis1week’.



(a)

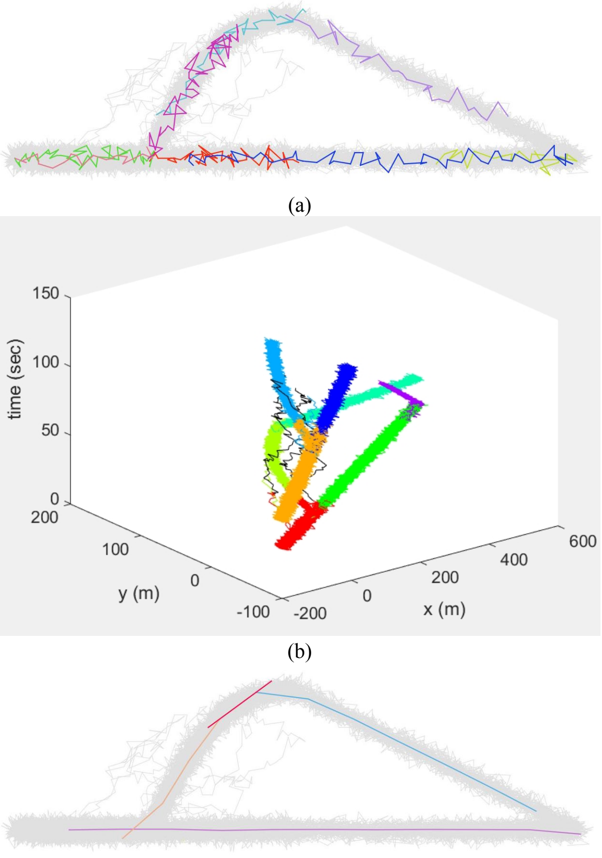
(b)

(c)

Figure 5. Visualization of the clusters' representatives provided by: S^{2}T-Clustering in (a) 2D and (b) 3D, (c) TRACLUS, when applied to a subset of SMOD consisting of 2 patterns.

First, we applied both S^{2}T-Clustering and TRACLUS [21] over a subset of SMOD that consists only of the trajectories that move throughout the whole lifespan of the dataset, thus limiting the ground truth to two clusters. In Figure 5(a) and Figure 5(c) we visualize only the representatives of each cluster, while in Figure 5(b) we provide a 3D illustration of the data used in the case of Figure 5(a). Note that S^{2}T-Clustering discovers the two clusters, while TRACLUS discovers several linear patterns; see Figure 5(a) vs. Figure 5(c).

Subsequently, we applied both S^{2}T-Clustering and TRACLUS to the entire SMOD, for which we have knowledge of the ground truth. In Figure 6(a) and Figure 6(c), we present the results of the S^{2}T-Clustering and TRACLUS, respectively. Moreover, in order to better comprehend the temporal dynamics of the dataset we provide a 3D illustration in Figure 6(b). According to this experiment, S^{2}T-Clustering effectively discovers all eight clusters (as well as the noisy sub-trajectories, depicted in black color in Figure 6(b)), thus S^{2}T-Clustering is not affected by the trajectories' shape, yielding an effective and robust approach for the discovery of linear and non-linear patterns. On the contrary, TRACLUS fails to identify the hidden ground truth in this SMOD due to the fact that it ignores the time dimension. Interestingly, TRACLUS discovers almost the same sets of representatives when applied to either a subset of or the entire SMOD; see Figure 5(c) vs. Figure 6(c).



(a)

(b)

(c)

Figure 6. Visualization of the clusters' representatives provided by: (a) S^{2}T-Clustering in (a) 2D and (b) 3D, (c) TRACLUS, when applied to the entire SMOD consisting of 8 patterns.

In order to evaluate the accuracy of our proposal in a quantified way, we further employed F-Measure in SMOD. In detail, we built 8 datasets, with the first consisting of the sub-trajectories of the first cluster only, the second consisting of the sub-trajectories of the first and the second cluster only, and so on, until the eighth dataset, which consisted of the sub-trajectories of all eight clusters; all eight datasets appeared in two variations: including or not the set of outliers. For each dataset, we applied S^{2}T-Clustering and calculated F-Measure; Figure 7 illustrates this quality criterion by increasing the number of clusters. It is evident that S^{2}T-Clustering turns out to be very robust, achieving always precision and recall values over 92.3%, while the outliers are always detected correctly.

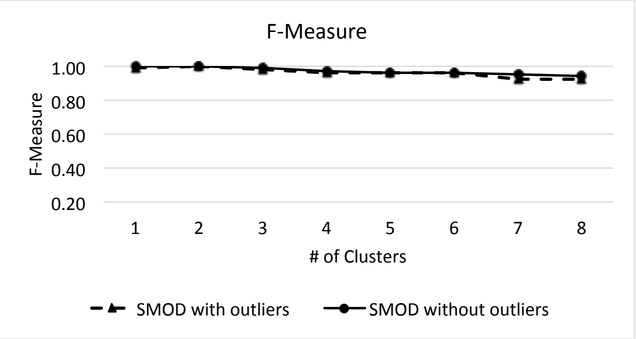


Figure 7. Quality of S^{2}T-Clustering w.r.t. number of clusters.

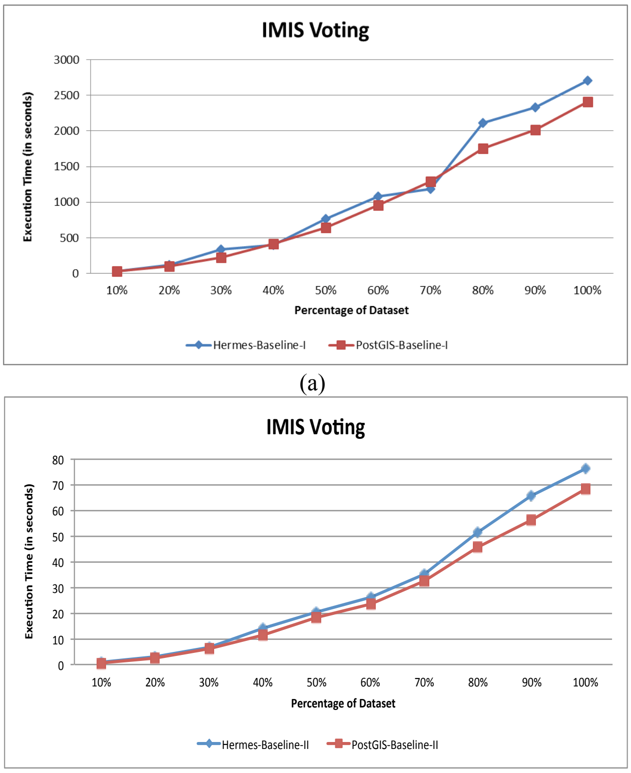
6.3 Efficiency and Scalability

In order to study the efficiency and scalability of our proposal we followed two competing approaches: Hermes@PostgreSQL [16], implemented according to the discussion in Section 5, vs. PostGIS extension of PostgreSQL that simulated the two baseline solutions presented in Section 5.1.

We have noticed that the implementation of the 3D-R-tree in PostGIS suffers from rounding errors because it uses 32-bit IEEE floating-point numbers to store the coordinates [35]. In our experiments we observed that the MBB of a trajectory or a segment was always enlarged due to this rounding, thus making the overlap query in PostGIS return more segments than our implementation. Since this made the comparison between the two systems unfair, we simulated PostGIS inside Hermes, in other words, also the baseline solutions were simulated inside Hermes (thus, making all solutions run under the same framework).

In the charts that follow, we denote the implementation of Baseline-I and Baseline-II solutions implemented both in Hermes and in PostGIS as {Hermes | PostGIS}-Baseline-{I | II}, i.e. four different implementations.

In particular, Figure 8 illustrates the execution time of the voting step for the IMIS dataset when varying the dataset size (i.e. the number of trajectories). Obviously, the two implementations present similar performance, with the PostGIS implementation performing slightly better mainly due to the fact that the size of index node entries in PostGIS (which uses 32-bit numbers for storing the temporal dimension) is slightly less than that of Hermes (which uses 64-bit numbers).



(a)

(b)

Figure 8. Comparing the performance of baseline solutions: (a) Baseline-I; (b) Baseline-II.

We repeated the same experiment with the GeoLife dataset and the results lead to similar conclusions, thus they are excluded due to space limitations.

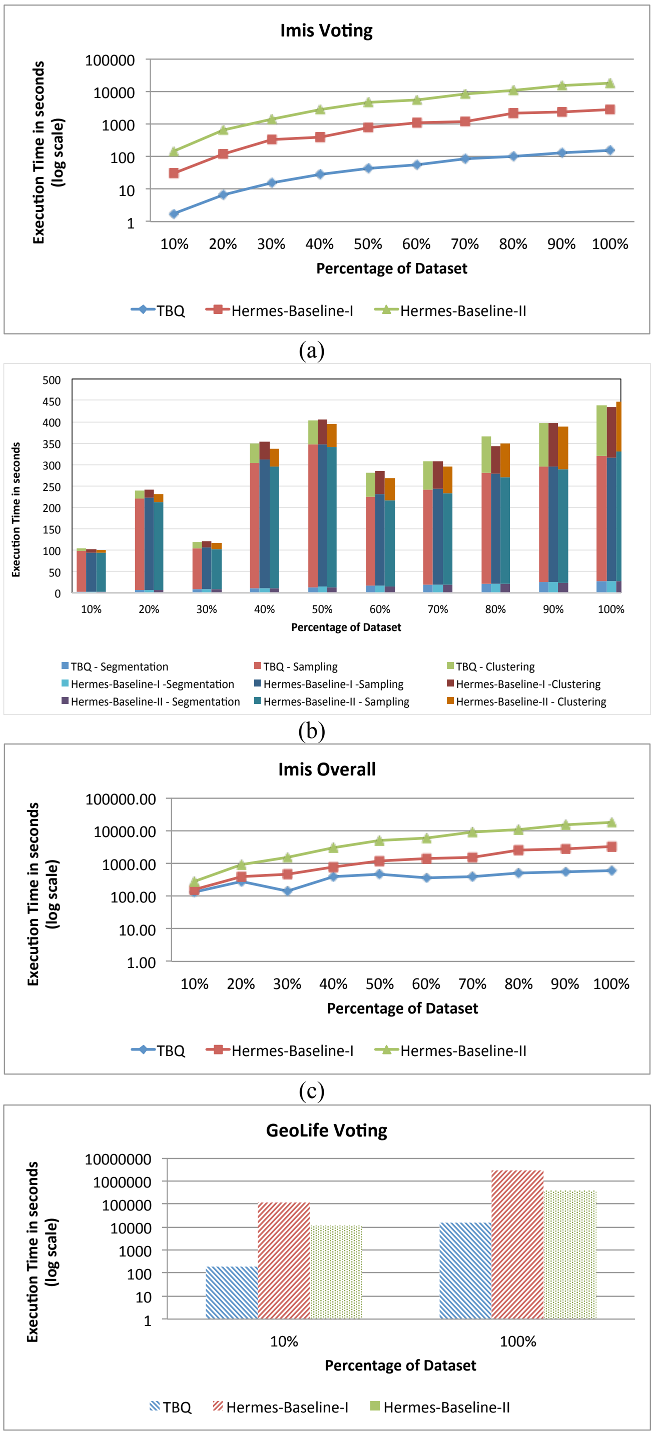
Based on the above results, in the remainder of the experimental study, the scalability study is conducted using the Hermes implementation of the algorithms. As illustrated in Figure 9(b), all three approaches (Baseline-I, Baseline-II and TBQ, presented in Section 5.1) perform similarly on the IMIS dataset as far as it concerns the segmentation, sampling and clustering steps of the algorithm (please note that y-axis is at log scale). The crucial difference is at the expensive voting step, where TBQ significantly outperforms the two baseline solutions by almost two orders of magnitude; this is illustrated in Figure 9(a) whereas in Figure 9(c) we present the accumulated processing time.

Due to the fact that the overall performance is dominated by the performance of the voting step, we further studied this step over the GeoLife dataset. As it can be observed in Figure 9(d), the behavior of the voting step of S^{2}T-Clustering over GeoLife is slightly different from that over IMIS. TBQ still outperforms both Baseline-I and Baseline-II solutions by several orders of magnitude, but in the case of GeoLife, Baseline-II outperforms Baseline-I. This can be explained by the fact that GeoLife consists of trajectories with significantly larger number of segments than IMIS (recall the statistics in Table 3). This fact leads Baseline-I to perform considerably more lookups in the index.

7. CONCLUSIONS

In this paper, we discussed the problem of sub-trajectory clustering and outlier detection in trajectory databases, aiming to take both space and time information into consideration. In particular, we proposed S^{2}T-Clustering that is novel not only because it solves the problem more effectively than the state-of-the-art (namely, TRACLUS), but also for an additional, quite important reason: our proposal is designed in-DBMS, i.e., it performs as a query operator in a real MOD engine over an extensible DBMS (namely, PostgreSQL in our current implementation). Having such functionality in their hands, data scientists are able to perform cluster analysis via simple SQL in real DBMS, where concurrency and recovery issues are taken into consideration. Moreover, our algorithm is boosted by an efficient index-based Trajectory Buffer Query (TBQ) that speeds up the overall process, resulting in a scalable solution, outperforming the state-of-the-art in-DBMS solutions supported by PostGIS by several orders of magnitude.

As a next step, inspired by the research agenda of the big data era, we plan to investigate real-time and incremental solutions, exploiting on modern in-memory DBMS architectures.



(a)

(b)

(c)

(d)

Figure 9. Step-by-step execution time of S^{2}T-Clustering: (a) voting over IMIS; (b) segmentation/sampling/clustering over IMIS; (c) overall over IMIS; (d) voting over GeoLife.

8. ACKNOWLEDGMENTS

This work was partially supported by project datACRON, which has received funding from the European Union's Horizon 2020 research and innovation Programme under grant agreement No 687591.

9. REFERENCES

[1] Almeida, V.T., Güting, R.H., & Behr, T. 2006. Querying moving objects in secondo. In Proceedings of MDM.

[2] Andrienko, G., Andrienko, N., Rinzivillo, S., Nanni, M., and Pedreschi D. 2009. A visual analytics toolkit for cluster-based classification of mobility data. In Proceedings of SSTD.

[3] Ankerst, M., Breunig, M. M., Kriegel, H.-P. and Sander, J. 1999. Optics: Ordering points to identify the clustering structure. In Proceedings of SIGMOD.

[4] Benkert, M., Gudmundsson, J., Hubner, F. and Wolle T. 2006. Reporting flock patterns. In Proceedings of ESA.

[5] Cadez, I. V., Gaffney, S., and Smyth, P. 2000. A general probabilistic framework for clustering individuals and objects. In Proceedings of KDD.

[6] Dodge, S., Weibel, R., and Lautenschütz, A.-K. 2008. Towards a taxonomy of movement patterns. Journal of Information Visualization. 7(3), 240-252.

[7] Ester, M., Kriegel, H.-P., Sander, J., Xu, X. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In Proceedings of KDD.

[8] Ferreira, N., Klosowski, J.T., Scheidegger, C.E., Silva, C.T. 2013. Vector Field k-Means: Clustering Trajectories by Fitting Multiple Vector Fields. In Proceedings of EuroVis.

[9] Frentzos, E., Gratsias, K., and Theodoridis, Y. 2007. Index-based most similar trajectory search. In Proceedings of ICDE.

[10] Gaffney, S., and Smyth, P. 1999. Trajectory clustering with mixtures of regression models. In Proceedings of KDD.

[11] Giannotti, F. and Pedreschi, D. 2008. Mobility, Data Mining and Privacy, Geographic Knowledge Discovery. Springer.

[12] Giannotti, F., Nanni, M. Pedreschi, D. Pinelli, F., Renso, C., Rinzivillo, S. and Trasarti, R. 2011. Unveiling the complexity of human mobility by querying and mining massive trajectory data. The VLDB Journal, 20(5): 695-719.

[13] Hadjieleftheriou, M., Kollios, G., Gunopulos, D. and Tsotras, V.J., 2006. Indexing Spatio-Temporal Archives, VLDB J., vol. 15, no. 2, pages 143-164.

[14] Hellerstein, J., Naughton, J. and Pfeffer, A. 1995. Generalized Search Trees for Database Systems. In Proceedings of VLDB.

[15] Hung, C.-C., Peng, W.-C., Lee, W.-C. 2015. Clustering and aggregating clues of trajectories for mining trajectory patterns and routes. The VLDB Journal, 24(2):169-192.

[16] Hermes@PostgreSQL MOD engine. URL: http://infolab.cs.unipi.gr/hermes/

[17] Jeung, H., Yiu, M. L., Zhou, X., Jensen, C., and Shen, H. T. 2008. Discovery of convoys in trajectory databases. In Proceedings of VLDB.

[18] Kalnis, P., Mamoulis, N., Bakiras, S. 2005. On discovering moving clusters in spatio-temporal data. In Proceedings of SSTD.

[19] Kollios, G., Gunopulos, D., Koudas, N., and Berchtold, S. 2003. Efficient biased sampling for approximate clustering and outlier detection in large datasets. IEEE Transactions on Knowledge and Data Engineering, 15(5):1170-1187.

[20] Kornacker, M. 1999. High-Performance Extensible Indexing. In Proceedings of VLDB.

[21] Lee, J.-G., Han, J., and Whang, K.-Y. 2007. Trajectory clustering: a partition-and-group framework. In Proceedings of SIGMOD.

[22] Li, Y., Bailey, J. Kulik, L. 2015. Efficient mining of platoon patterns in trajectory databases. Data & Knowledge Engineering, 100(PA):167-187.

[23] Li, Z., Ding, B., Han, J., Kays, R. 2010. Swarm: Mining relaxed temporal moving object clusters. In Proceedings of the VLDB Endowment 3(1-2):723–734.

[24] Li, Z., Lee, J.G., Li, X., Han, J. 2010 Incremental clustering for trajectories, In Proceedings of DASFAA.

[25] Li, Z., Ji, M., Lee, J.-G., Tang, L.-A., Yu, Y., Han, J., Kays, R. 2010. MoveMine: mining moving object databases. In Proceedings of SIGMOD.

[26] Nanni, M., and Pedreschi, D. 2006. Time-focused clustering of trajectories of moving objects. Journal of Intelligent Information Systems, 27(3):267-289.

[27] Nanopoulos, A., Theodoridis, Y., and Manolopoulos, Y. 2006. Indexed-based density biased sampling for clustering applications. Data and Knowledge Engineering, 57(1):37-63.

[28] Panagiotakis, C., Pelekis, N., Kopanakis, I., Ramasso, E., and Theodoridis, Y. 2012. Segmentation and sampling of moving object trajectories based on representativeness. IEEE Transactions on Knowledge and Data Engineering, 24(7):1328-1343.

[29] Pelekis, N. and Theodoridis, Y. 2014. Mobility Data Management and Exploration. Springer.

[30] Pelekis, N., Andrienko, G., Andrienko, N., Kopanakis, I., Marketos, G., Theodoridis, Y. 2011. Visually Exploring Movement Data via Similarity-based Analysis. Journal of Intelligent Information Systems, 38(2):343-391.

[31] Pelekis, N., Frentzos, E., Giatrakos, N., and Theodoridis, Y. 2008. HERMES: Aggregative LBS via a trajectory DB engine. In Proceedings of SIGMOD.

[32] Pelekis, N., Kopanakis, I., Kotsifakos, E., Frentzos, E. and Theodoridis, Y. 2011. Clustering uncertain trajectories. Knowledge and Information Systems, 28(1):117-147.

[33] Pelekis, N., Panagiotakis, C., Kopanakis, I., and Theodoridis, Y. 2010. Unsupervised trajectory sampling. In Proceedings of ECML-PKDD.

[34] Pfoser, D., Jensen, C.S., and Theodoridis, Y. 2000. Novel approaches to the indexing of moving object trajectories. In Proceedings of VLDB.

[35] Ramsey, P. (on behalf of PostGIS), personal communication.

[36] Tang, L.A., Zheng, Y., Yuan, J., Han, J., Leung, A., Hung, C. and Peng, W. 2012. Discovery of Traveling Companions from Streaming Trajectories. In Proceedings of ICDE.

[37] Tang, L.A., Zheng, Y., Yuan, J., Han, J., Leung, A., Peng, W. and Porta, T. L. 2012. A Framework of Traveling Companion Discovery on Trajectory Data Streams. ACM Transactions on Intelligent Systems and Technology, 5(1).

[38] Theodoridis, Y., Vazirgiannis, M. and Sellis, T. 1996. Spatio-Temporal Indexing for Large Multimedia Applications. In Proceedings of ICMS.

[39] Wang, S., Wu, L., Zhou. F. Zheng, C., Wang, H. 2015. Group Pattern Mining Algorithm of Moving Objects' Uncertain Trajectories. International Journal of Computers, Communications & Control, 10(3):428-440.

[40] Wu, F., Lei, T.K.H., Li, Z. Han, J. 2014. MoveMine 2.0: mining object relationships from movement data. In Proceedings of VLDB.

[41] Xu, H., Zhou, Y., Lin, W., Zha, H. 2015. Unsupervised Trajectory Clustering via Adaptive Multi-Kernel-based Shrinkage. In Proceedings of ICCV.

[42] Yuan, G., Sun, P., Zhao, J., Li, D. Wang, C. 2016. A review of moving object trajectory clustering algorithms. Artificial Intelligence Review, 1-22.

[43] Zhang, T., Ramakrishnan, R., and Livny, M. 1996. Birch: An efficient data clustering method for very large databases. In Proceedings of SIGMOD.

[44] Zheng, K., Zheng, Y., Yuan, N. J., Shang, S. 2013. On Discovery of Gathering Patterns from Trajectories. In Proceedings of ICDE.

[45] Zheng, K., Zheng, Y., Yuan, N. J., Shang, S., Zhou., X. 2014. Online Discovery of Gathering Patterns over Trajectories. IEEE Transaction on Knowledge and Data Engineering, 26(8):1974-1988.

[46] Zheng, Y. 2015. Trajectory Data Mining: An Overview. ACM Transactions on Intelligent Systems and Technology, 6(3).

[47] Zheng, Y., Xie, X., Ma, W.-Y. 2010. GeoLife: A Collaborative Social Networking Service among User, location and trajectory. IEEE Data Engineering Bulletin, 33(2):32-40.

Powering Archive Store Query Processing via Join Indices

Joseph Vinish D'silva[1], Bettina Kemme[1], Richard Grondin[2], and Evgueni Fadeitchev[2]

[1]School of Computer Science , McGill University , joseph.dsilva@mail.mcgill.ca , kemme@cs.mcgill.ca
[2]ILM Development , Informatica , rgrondin@informatica.com , efadeitchev@informatica.com

ABSTRACT

In recent years, the industry landscape surrounding data processing systems has been significantly impacted by Big Data. Core technology and algorithms for data analysis have been adjusted and redesigned to handle the ever increasing amount of data. In this paper we revisit the concept of *join index*, a base mechanism in relational DBMS to support the expensive join operator, and analyze how it can be effectively integrated and combined with other mechanisms widely deployed for large-scale data processing. In particular, we show how the data store *Informatica IDV*, originally designed to facilitate backup and archival of application data, can benefit from join indices to give fast SQL-based access to archival data for discovery purposes. Informatica IDV supports both horizontal and vertical partitioning – two mechanisms that are widely used in modern data stores to speed up large-scale data processing. However, this requires us to reexamine join index design and usage. In this paper, we propose a scalable, partitioned, columnar join index that supports parallel execution, ease of maintenance and a late materialization query processing approach which is efficient for column-stores. Our implementation based on *Informatica IDV* has been evaluated using a TPC-H based benchmark, showing significant performance improvements compared to executions without join index.

CCS Concepts

•Information systems → Join algorithms;

Keywords

join indices; predicate evaluation; archive stores;

1. INTRODUCTION

The past decade has seen a surge in data analytics, primarily driven by Big Data. Gartner predicts the market forecast for BI & Analytics sector to reach $16.9 billion in 2016, an increase of 5.2 percent from 2015 [6]. Falling disk storage prices [12] and off-the-shelf hardware costs in general have resulted in an increasing number of organizations taking the Big Data leap. The unprecedented abundance of data and the demand to process them efficiently and economically has resulted in various emerging trends.

Big Data frameworks like the Hadoop ecosystem, a prominent technology to process large amount of unstructured data, are engineered to run on clusters that can be easily built from off-the-shelf commercial hardware without needing any specialized and costly components. They also make fault tolerance concepts, such as persisting intermediate results, stateless worker tasks, shared and replicated storage etc., a fundamental part of their design. Most Big Data applications are centered around use cases that have very little update to existing data and hence, can exploit storage structures optimized for appends. Data is usually partitioned horizontally, allowing the framework to process the various data partitions independently in parallel.

RDBMS vendors, on the other hand, have started feeling the pinch to reduce the amount of I/O incurred during query processing as the data sizes grew. A fundamental reason for the high I/O has been due to the row-based storage of data, that often results in reading a lot of attributes from disk that are not required. This is where column stores have found their resurgence, as they store each column in separate blocks in the disk, often referred to as vertical data partitioning. [1] and [10] demonstrate that column-stores perform better than row stores for analytical queries. These observations have forced the leading row-based RDBMS vendors to incorporate many features of column stores [22]. However [1] concludes that such optimizations on row-stores still fall short of the column-store performance. The time point the resultset of a query is *materialized* has a particular impact in the performance. Row stores traditionally use *early materialization*, i.e., building the final resultset's attributes as early as possible whenever they access a potentially relevant row for the first time (even if the row might be later disregarded), while [2] shows that column-stores benefit from *late materialization*, where the output columns for a tuple are only retrieved when it is ensured that the tuple qualifies all predicates, leading to a significant reduction in I/O.

A further player in the large-scale data processing domain are *Data Warehouses (DW)*. Although originally conceived to process analytical queries on historical data, the need for more up-to-date information in the form of real-time Business Intelligence and event-driven processing has increased the complexity of the DW systems both in terms of software and hardware. The latter, for instance, is often character-

Series ISSN: 2367-2005

10.5441/002/edbt.2017.85

ized by large main memory, multi-processor servers attached to fast, reliable storage such as SSDs. However, the resulting higher price tag might be prohibitive for many application domains, given that many DW vendors consider storage size as a major factor for pricing their market offerings.

Therefore, organizations started looking whether their *archive stores*, that were traditionally seen only as an infrastructure to facilitate backup and retirement of application data that has some retention requirements, could be leveraged to perform data discovery. Given that the source systems for archive stores are predominantly RDMBS applications, they inherit the semantic structure and data quality from the source systems - a principal difference with typical Big Data systems of today that are predominantly tailored to process unstructured data. As a result, it is more natural for archive stores to offer the familiar SQL query interface, and behave more like an RDBMS – which appears attractive as RDBMS have shown to outperform BigData systems like MapReduce when it comes to performing relational operations [18]. Thus, the potential for query optimization, in particular, when the archive store follows a column-store approach, is high.

Additionally, as data is typically appended as chunks, and later seldom updated, archive stores have the potential to benefit from horizontal partitioning in a similar way as Big Data Frameworks, facilitating shared storage and stateless computing tasks design, and allowing for parallel and fault-tolerant query processing. Therefore, building on the lessons from Big Data systems and Column-stores, we can observe that archive stores, in particular when they deploy both vertical and horizontal partitioning, stand to gain by taking a leaf from both of these technologies.

However there is an important part of query processing in relational systems that is also very costly - *joins*! Among other techniques, some RDBMS have employed an auxiliary data structure known as *join index* to address join performance. A join index represents a fully pre-computed join between two or more relations by storing some form of source table row identifier for each resultset tuple [16]. While join indices can occur significant maintenance overhead when the source tables change frequently, they are an attractive proposition for archive stores where data is typically appended incrementally and existing data might only be changed in batches, thus making it possible to do efficient batch maintenance on join indices. Also, and as we will see in this paper, appropriately designed join indices lend themselves well to partitioned data, thus providing great potential for scalability.

Therefore, in this paper, we hypothesize that join indices can be highly beneficial for archive stores and develop an implementation for a columnar, highly-scalable, archive data store, Informatica IDV, analyzing carefully how data distribution and the columnar architecture affects the join-index design. Our approach naturally follows the horizontal partitioning approach deployed in IDV, and performs join index maintenance on a partition basis. We leverage the existing columnar storage structure by persisting the join indices as special system tables whose columns are rowids of tuples of the different relations that join. We also implement new query execution workflows that can utilize the join indices which is in concordance with the way partitions are processed currently in IDV, facilitating parallel processing of join queries. Furthermore, we develop a new methodology of evaluating selection predicates which addresses the costs associated with redundant predicate processing. Finally, we implement a late materialization strategy where projection attributes of matching tuples are retrieved as late as possible to take advantage of the columnar storage.

Our tests using a TPC-H[1] based benchmark with different queries and database configurations show conclusively that using our join index does indeed offer significant performance improvements on join query processing compared to non-join index based joins in terms of execution times and CPU, I/O and memory usage.

In short, our paper makes the following contributions.

- A join index design for a partitioned, scalable and columnar database leveraging the existing storage structures for simplified implementation and maintenance.
- A holistic query execution strategy with improved selection predicate processing that avoids redundant evaluations of selection predicates in multi-partition joins.
- A late materialization based approach for generating the output result leveraging on the columnar storage.
- A detailed analysis of the performance of our join index implementation using the TPC-H benchmark suite.

2. BACKGROUND AND RELATED WORK

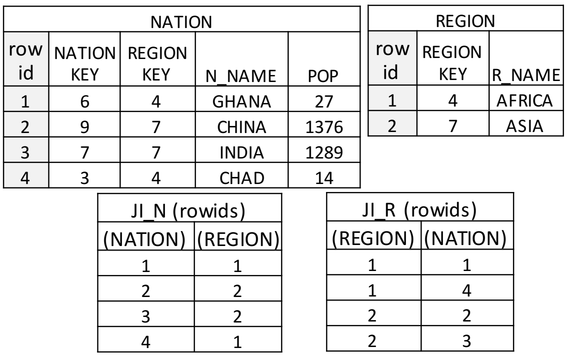
Join is one of the most fundamental – and one of the most costly, operations in relational query processing. Most common is the equi-join where the join attributes are the primary key and foreign key of the respective relations to be joined. A join can be defined over multiple relations whereby an N-way join can be computed as a series of $N-1$ 2-way joins. Furthermore, complex SQL queries typically combine joins with *selection predicates* on individual attributes of the participating relations (`WHERE` clause of SQL statements), and have the result set only *project* on a subset of all possible attributes (`SELECT` clause of SQL statements). Thus, it is not only crucial to find efficient ways of executing the join operations themselves [16] but also to integrate join execution with selection and projection tasks.

There exists a variety of physical join mechanisms following different query processing strategies [15, 7], and targeting various data characteristics and DBMS architectures. In general, join mechanisms can be classified as *(i)* not depending on specialized data structures - such as nested join, sort-merge join, hash join algorithms and their variants; *(ii)* or depending on specific data structures such as indices that need to be built and maintained. The most prominent join-specific data structures can be broadly classified as *links* [9, 19], *materialized views* [20] and *join indices* [23, 5, 21, 17, 14]. For the sake of brevity, we will confine our background discussions to some of the fundamental approaches of join indices and query processing, as is relevant to this paper.

2.1 Join Index

Join indices in its current familiar form were defined by Valduriez in [23] as a special relation that represents the abstraction of the join of two relations. Though other variants [5, 21, 17, 14] exist, the primary design concept of join index remains more or less the same. A join index for two

[1] http://www.tpc.org/tpch/spec/tpch2.15.0.pdf



NATION				
row id	NATION KEY	REGION KEY	N_NAME	POP
1	6	4	GHANA	27
2	9	7	CHINA	1376
3	7	7	INDIA	1289
4	3	4	CHAD	14

REGION		
row id	REGION KEY	R_NAME
1	4	AFRICA
2	7	ASIA

JI_N (rowids)	
(NATION)	(REGION)
1	1
2	2
3	2
4	1

JI_R (rowids)	
(REGION)	(NATION)
1	1
1	4
2	2
2	3

Figure 1: Join index impl. according to Valduriez

relations is a relation with two attributes where each tuple of the index represents a pair of tuples of the base relations that join according to the join criteria. For an equi-join of two relations R and S, the join index relation JI can be represented using the definition adapted from [23] as

$JI = \{ (r_i[rowid], s_j[rowid]) \mid r_i[att_R] == s_j[att_S] \}$

Where r_i and s_j are tuples from the relations R and S respectively, att_R and att_S are the attributes over which the join is defined, and *rowid* is a database generated surrogate that is used to uniquely identify the tuples within the particular relations. A join index can also be regarded as a special form of a materialized view [17] as it represents a pre-computed join between tables with only their rowid attributes materialized.

A join index needs further processing to build the result-set by accessing the required attributes of the selected tuples from the underlying tables. This operation will have to be performed with significant efficiency, otherwise any performance advantage of having the joining tuples pre-computed will be lost. If the join index has to be used in combination with tuple selection based on either of the relations, Valduriez suggested that two copies of join index be maintained with each one *clustered* on the rowids of one of the relations [23]. Fig. 1 shows an example of two join indices built between `NATION` and `REGION`, one clustered on `NATION` (`JI_N`) and the other one (`JI_R`) clustered on `REGION`. Now assume the following join query over these two relations that additionally contains a selection predicate on `REGION`.

```
SELECT *
FROM REGION JOIN NATION
ON REGION.REGION_KEY = NATION.REGION_KEY
WHERE R_NAME = 'ASIA'
```

For this query, the `REGION` table and the index `JI_R` can be scanned sequentially and in tandem. The scan on `REGION` will determine that only the rowid 2 of `REGION` qualifies the selection predicate, and thus, only the last two rows in `JI_R` are relevant. Therefore, only the `NATION` tuples with rowids 2 and 3 will be retrieved to build, together with the already loaded tuple of `REGION` with rowid 2, the result set.

[17] describes a bitmap-based join-index that is suited for star schema joins. In this approach, a join index is created such that for each record in the dimension table, a bit string that corresponds to the length of the fact table is stored in the join index (i.e., the number of bits equals the number of rows in the fact table). Individual bits on the bit string map to the rowids of the fact table. A bit is set if that fact table row joins with the row corresponding to the dimension table entry.

[24] also proposes a join index that is suited for a star schema. It applies a hybrid storage model in which the fact table is maintained as a row store, whereas frequently accessed dimension tables are stored in a columnar fashion. The fact table is transformed into a join index by replacing the dimensional attributes stored in the fact table with references to the corresponding tuple in the dimension table.

[5] proposes a composite attribute and join index which is a variation of the concept of links described in [9]. The index structure, termed B_c-tree is based on the concept of a B^+-tree. The leaf nodes of the B_c-tree contain references (in principal, pointers to the physical locations) to all the tuples in the database which share the same data values of a common domain. Thus, the structure serves as a secondary index on an attribute as well as a multi-way join index. B_c-tree can also be used to enforce integrity constraints, since the values of the domain are stored as part of the tree structure. Joins are performed by accessing the tuples via the references stored in each of the leaf nodes. For joins without additional selection, the search is performed by means of a sequential traversal of the leaves of the B_c-tree [5]. Compared to the regular join index [23], this implementation can support multiple joins based on the same attribute simultaneously.

Comparative studies of the performance of join indices, materialized views and join algorithms have been described in [3] and [16]. [3] concluded that the method of choice to implement joins was dependent on various environmental characteristics like join selectivity[2], main memory availability, volatility of the attributes of base relations etc. Although there are various optimizations of join algorithms, it has been established that in most scenarios, join indices can provide better performance compared to other join mechanisms in traditional RDBMS [13].

Little work on join indices exists outside the scope of row-based RDBMS. An exception is [4] where join indices are created on the fly during query processing for the column-based DBS MonetDB. The approach has some similarities to ours due to both being based on a column storage which, as we discuss below, brings advantages in terms of late materialization. But our approach is more general as it also considers horizontal partitioning, carefully integrates with selection operators, and stores join indices persistently.

2.2 Query Processing Approaches

In this section, we point at two fundamental approaches related to scalability and performance that are of interest for us in our join index design, namely operator pipelining and materialization techniques.

2.2.1 Pipelined Operation for Efficiency

Most RDBMS support pipelined query processing, where operators pass their output (often using intermediate buffers) as they are produced to the next operator in the processing step. This improves performance as operators can work in parallel producing results faster, and in some scenarios providing the first rows while the query is still processing the remaining records. But this tightly coupled query processing

[2] The *selectivity factor* is defined as the ratio of the number of result tuples of a join operation to the number of tuples in the Cartesian product of the underlying relations.

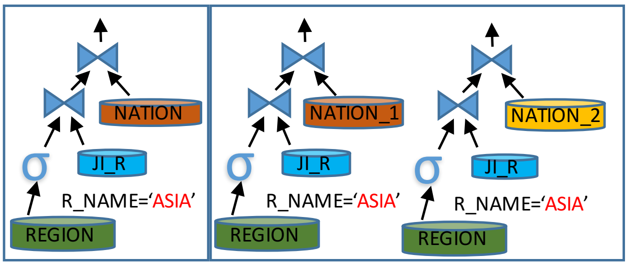


Figure 2: Impact on pipelining due to horizontal partitioning

approach comes at a cost to fault tolerance. A failure in one of the operator tasks can result in having to reprocess the entire query workflow. With traditional RDBMS this was not a significant issue, as with small number of computing nodes, the rate of failures were very low to be of concern.

Looking more closely at the execution of the SQL join query given in the previous section, we can see the pipelined approach, depicted on the left side of fig. 2. The selection operator reads records from `REGION` and pipelines the qualifying tuples to the operator that looks-up the join index `JI_R`. This operator produces the matching rowids of `NATION` in its output, which are further pipelined to an operator that reads the corresponding tuples from the `NATION` table.

However, this pipelined approach does not necessarily scale well on a horizontally partitioned system. If `NATION` had two partitions, then, if we want to achieve parallelism via scale-out, we want to process both partitions at the same time. A trivial design approach is to have each of the partitions to be joined separately with `REGION` table in a pipelined fashion. However, this will require the selection operator to be applied twice on the same data of `REGION` as shown on the right side of fig. 2.

In fact, if tables $T_1, T_2, \ldots T_n$ are joined in that order, with each of them having $p_1, p_2, \ldots p_n$ number of partitions, then the number of duplicate selection predicate evaluations for a table T_i has an upper bound of $(\prod_{j=1}^{i} p_j) - p_i \ : i > 1$. This can translate to unnecessary I/O in a system with large number of partitions. We will discuss how we tackle this effectively in our query processing approach in the next section.

2.2.2 *Materializing Strategies*

Once determined in which order operators are executed in the execution tree the question arises what information is exactly transmitted from one operator to the next. If a system that stores all attributes of a row in a single chunk (row-based storage), it makes sense to retrieve all attributes of a row that are needed for further processing the first time any operator accesses this specific row and include them in the data that is moved to the next operator. This *early* materialization that grabs all attributes that might be potentially useful in the first disk read can reduce the overall I/O costs, as later steps, for example when generating the final attributes to be returned, do not need to read the tuple again, which might lead to additional I/O.

As an example, let's have a look at the query

```
SELECT N_NAME, POP
FROM REGION JOIN NATION
ON REGION.REGION_KEY = NATION.REGION_KEY
WHERE R_NAME = 'ASIA' AND N_NAME LIKE 'C%'
```

For this query, the `NATION` record for `CHAD` fulfills the selection operator, and thus, early materialization will retrieve the `N_NAME` and `POPULATION` attributes and forward them to the join operator. However, this tuple will not find a matching `REGION` tuple and thus, will be eliminated by the join. Thus, the I/O cost for reading the attributes from disk and forwarding them to the next operator is an overhead that we incur in our efforts to avoid re-reading the same data disk blocks later to generate the output list. Holistically, any data that is read from the disk, but later not used for query processing (because the tuple was discarded at a later step), leads to wastage of resources.

It is in this context where column-stores, with their late materialization approach, provide better results. In a columnar model, each of the attributes is stored separately in a different data block (or set of data blocks). Hence, when looking for nations with `N_NAME LIKE 'C%'`, the selection operator only needs to read the data blocks associated with `N_NAME`. It can then produce the rowids that qualify the selection in its output and transmit this set to the join operator. Similar approach holds for the selection operation to determine the set of rowids with `R_NAME = 'ASIA'`. The join operator can then determine the join index tuples that contain rowids from both the sets. This can be then consumed by a result generator which can lookup the attributes required in the output and construct the output tuples. Albeit a bit more complex than early materialization, we can see that with large scale data processing systems, this avoids wasting precious I/O. For example, in the example above, the data blocks for the attributes `NATION_KEY` and `REGION_KEY` do not need to be read at all, and neither does the `POP` attribute value for the record with `N_NAME` equal `CHAD`.

Work done in [2, 1] demonstrates how late materialization strategy provides performance boost for column-stores over the early materialization based approach of traditional row-stores. In summary, an advantage of column-stores is that they are naturally suited for late materialization as all the columns are stored in separate data blocks [1]. Thus, column-stores can perform joins and selections by reading just the columns required for the joins/selections without fetching any other attributes. And then, for the final projection, only the projection attributes from matching tuples need to be retrieved. This makes them I/O efficient compared to row-stores that always retrieve the entire record upon the first access.

2.3 IDV in a Nutshell

Informatica IDV serves as a relational archive store for Informatica's ILM Application suite, and provides access to the archived data via standard SQL interfaces. The data store follows a distributed architecture offering parallel execution of SQL queries. IDV provides columnar storage (vertical partitioning) as well as horizontal partitioning. New data gets appended as additional (horizontal) partitions in immutable file structures[3] [11]. The data files follow a pro-

[3] IDV supports logical delete by storing information about deletions as extensions to the partition as well as facilitates

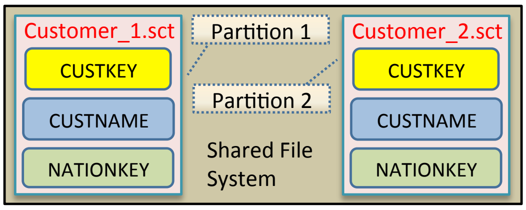


Figure 3: Database storage layout

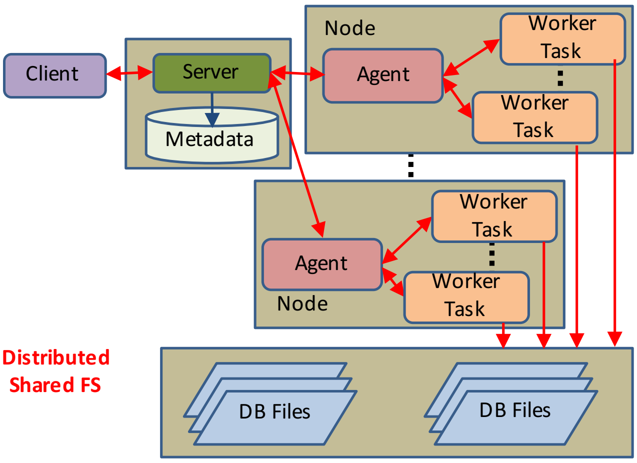


Figure 4: IDV high level database architecture

prietary format called Segment Compacted Table (SCT) and are usually stored in shared/distributed filesystems, decoupling the storage and computation aspects of the database. An abstract depiction of this storage structure is shown in fig. 3.

The high level architecture of IDV (fig. 4) bears a lot of resemblance to Big Data frameworks. Clients interact with the database *server* which generates *tasks* to execute query plans and places them in the execution queue. The server uses the metadata (data location and partitioning, data statistics, etc.) stored locally to determine the execution plans. The *agent* processes run on computing nodes. They pick up tasks and spawn *worker tasks* to execute them. The worker tasks are stateless by design and read data from the shared storage and persist the output back to the shared storage. This facilitates multi-step query processing, parallelism, fault tolerance, etc., quite similar to Big Data frameworks like MapReduce. The final output is sent back to the client via the agents and server, the latter also consolidating results from various tasks.

The conventional join query processing in IDV follows a pipelining approach where N-table joins are executed as a sequence of 2-table joins and selections on tables are performed before the tuples are fed into the join operator. By default, IDV uses a merge join. For instance, assuming a 3-table join over tables T_1, T_2, T_3 with having p_1, p_2, p_3 partitions respectively, there is a worker task for each combination of partitions of T_1 and T_2 (i.e., $p_1 * p_2$ worker tasks). Each of these worker tasks applies the selection predicates relevant to the partition(s) it is working on, constructing a memory resident bit vector called *Tuple Selection Vector* (TSV) [8]. TSVs indicate which rows qualify from a partition by turning on the corresponding row's bit position and are stored in compressed format to reduce memory footprint. The TSV approach is conceptually similar to vectorized query processing described in [1]. The worker task then uses the information from the TSVs to retrieve the remaining attributes required to perform the actual join and generate the result set. The resulting tuples then build an output partition that is one of the input partitions for the next set of worker tasks. The second set of worker tasks join the partitions generated by the first join with the partitions of T_3 again building TSVs for T_3 as needed. There is a total of $p_1 * p_2 * p_3$ such worker tasks.

As selections are performed in a pipelined fashion by the worker tasks that also do the join, and every partition joins with many other partitions, there is a redundant execution of selections as discussed in section 2.2. We will see how to avoid this in our join implementation. Furthermore, IDV currently performs early materialization, which is not necessarily beneficial and can be avoided in column-stores.

3. COLUMNAR JOIN INDEX

In this section we present the design and implementation of our join index that works together with IDV's column-based partitioning to support late materialization and horizontal partitioning to support parallel computation. It also clearly separates the selection operation from the join in order to avoid redundant computation. Our design does not only support 2-table join indices but arbitrary N-table join indices. The idea is to create an N-table join index whenever the application has many queries that join these N tables. Additionally, our N-table join index does not only serve queries that join exactly these N tables but also potentially queries that join a subset or a superset of these tables. Furthermore, as IDV updates the data on a partition basis, the index join maintenance can be done incrementally, so that the addition or the modification of a partition only requires a partial regeneration of the join index.

3.1 Join Index Creation

We create join indices in a partitioned fashion by creating a join index partition for each combination of base table partitions. Fig. 5 portrays the structure of the join index for a three-table, many-partition join based on a subset of the TPC-H schema, consisting of relations `REGION`, `NATION` and `CUSTOMER` that are connected through foreign key relationships. The join index has a total of 6 partitions. Our IDV based implementation stores each of the join index partitions in a columnar fashion as special system tables, making use of the existing database storage APIs. An important advantage of maintaining the join index in partitioned format is that each join index partition can be processed by a different worker task, providing ample opportunity for parallelism.

Number of Join Index Partitions. In the general case, assuming a N-table join index should be created for base tables T_1, T_2, $\ldots T_n$, with each of them having p_1,p_2, $\ldots p_n$ number of partitions, respectively, we will create potentially $\prod_{j=1}^{n} p_j$ join index partitions. There will be $\prod_{j=1,j\neq i}^{n} p_j$ join index partitions mapped to a given partition of T_i.

rebuilding the whole partition to purge them, but this does not have significant bearings in our approach and hence will not be discussed in depth.

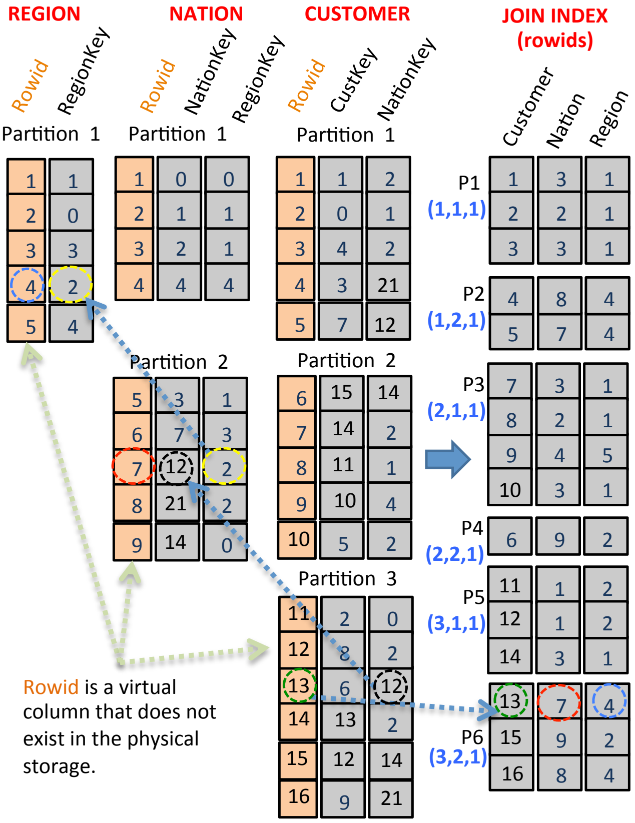


Figure 5: 3-Table Join Index Example

However, in many scenarios the number of actual join index partitions will be potentially much lower than this theoretical upper bound. This is because, in many real world scenarios, many combinations of source table partition joins will not yield any records in the output due to the associative nature of data in partitions across related tables. For example, consider an ORDER table, and a related LINEITEM table, which lists for each order in the ORDER table the items purchased under this order. As partitions of both ORDER and LINEITEM tables are added to the system as the orders are created, e.g., on a per-day basis, all ORDER and LINEITEM records with the same ORDERDATE will be in the same partitions. Thus, there will be a 1 : 1 mapping between the partitions of both the tables, and all tuples of a partition of LINEITEM will only match with the one corresponding partition of ORDER. Any joins between the rows in partitions with different values for ORDERDATE will yield no output.

Join Index Maintenance. We can maintain the join index in an incremental fashion. Whenever a new partition is added to a table, say to CUSTOMER, only this partition has to be joined with all existing partitions of the other tables to create new join index partitions. The existing join index partitions are not affected. The removal of a partition has as effect the removal of the join index partitions that are involved with the deleted partition.

As each combination of source table partitions is mapped to a different join index partition, it reduces the storage requirements by having to store only the rowid and not the partition numbers, as the later can be captured as metadata.

3.2 Query Processing Using Join Index

We modified IDV's query processing approach presented in section 2.3 to execute N-table join queries using our join indices. The modified query processing workflow consists of two steps as depicted in fig. 6. In the first step, all predicate selections declared in the query are performed. This involves generating and persisting TSVs for each partition of the participating relations. This can be performed in parallel. Once the TSV generation step is completed, the actual join index query processing takes place. This step is also capable of parallel execution so that multiple join index partitions are processed in tandem. These two steps constitute the primary components of the new join query workflow, and are independent of the number of tables involved in the join, contrary to the regular join workflow which involves $N-1$ steps. In the following, we discuss in detail these two steps, and how they differ from the original query processing.

3.2.1 *Evaluating selection predicates*

The original query execution process performed joins by joining each partition of the first relation with each partition on the second relation. Evaluating selection predicates was tightly coupled with each of these partition-partition joins. That is, for two partitions T_{1i} and T_{2j} of tables T_1 and T_2 to be joined where there is a selection predicate on the tuples of T_1, the IDV worker task performs the selection predicate evaluation by retrieving the column(s) on which the condition is specified, testing for validity, and constructing the TSV for T_{1i}. An important drawback of the current implementation is that the TSVs are not shared between worker tasks, even if they are working on the same table partition, evaluating the same selection predicates. As a single table partition is involved in as many joins as there are partitions of the joining (intermediate) table, this introduces a lot of redundant TSV generations, at times depending on the nature of the joins in the query and the predicates applied. I.e, if p_1 is the number of partitions in table T_1 and it is being joined with a (intermediate) table T_2 with p_2 partitions, then we are looking at a possible $p_1 \times (p_2 - 1)$ redundant TSV evaluations for the partitions of table T_1.

In general, it can be shown that if tables $T_1, T_2, \ldots T_n$ are joined in that order, with each of them having p_1,p_2, $\ldots p_n$ number of partitions, then the redundant TSV evaluations of the i^{th} table T_i has an upper bound of:

$$\le \begin{cases} (\prod_{j=1}^{2} p_j) - p_1 & : i = 1 \\ (\prod_{j=1}^{i} p_j) - p_i & : i > 1 \end{cases}$$

When using a join index for join query processing, we still need to evaluate the selection predicates, as a join index in general is built with just an equijoin between the relations and contains entries for all joining tuple combinations. In order to achieve maximum parallelism, we need to employ a worker task to process each join index partition. Following the current approach on tight integration of selection predicate evaluation with the join process will only result in aggravating an already existing problem of redundant TSV evaluations. In general, each of the tables T_i would contribute to $p_i \times (\prod_{j=1, j \neq i}^{n} p_j - 1)$ redundant TSV evaluations towards the join query. The total number of redundant TSV

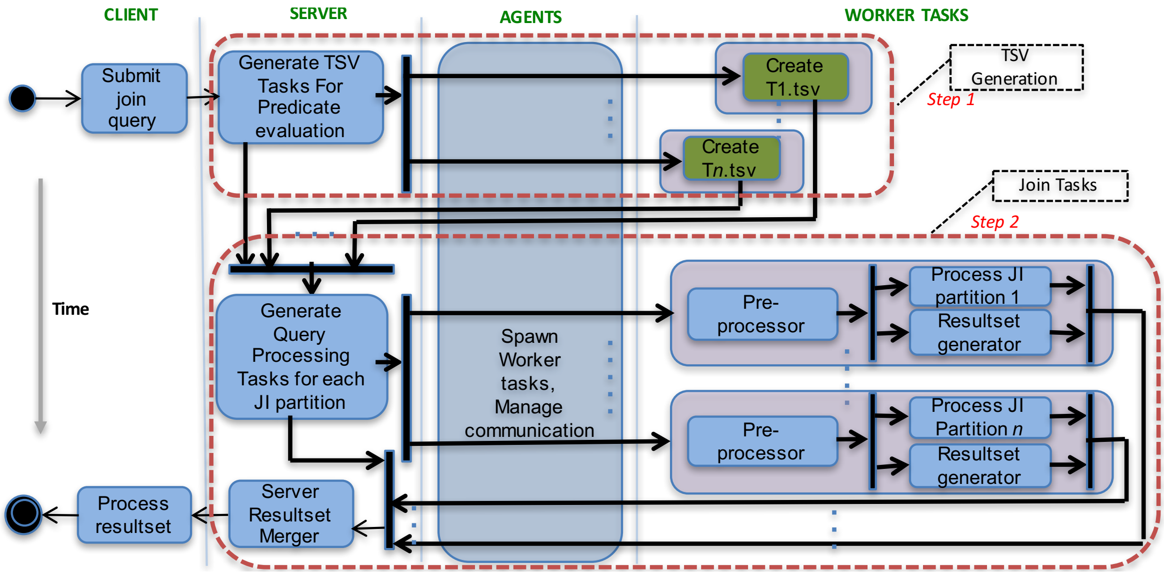


Figure 6: Activity diagram for query processing workflow using join index

evaluations for the query can be given by.

$$\sum_{i}^{n}(p_i \times (\prod_{j=1,j\neq i}^{n} p_j - 1)) = n(\prod_{j=1}^{n} p_j) - \sum_{j=1}^{n} p_j$$

In order to overcome this design predicament, we decided to separate the TSV generation step, which was tightly integrated with the join processing, into a separate step, and to have the TSVs persisted in shared disks for reuse so that the same partitions undergo only one selection predicate evaluation. This method avoids all the redundant TSV generations and generates the bare minimum number of TSVs required, which is the same as the total number of partitions across all the participating tables in the join, i.e., $\sum_{j=1}^{n} p_j$. Another advantage of this strategy is that the TSVs for all the partitions of all the tables can be processed in parallel, as they are independent of each other, thereby reducing the overall processing time.

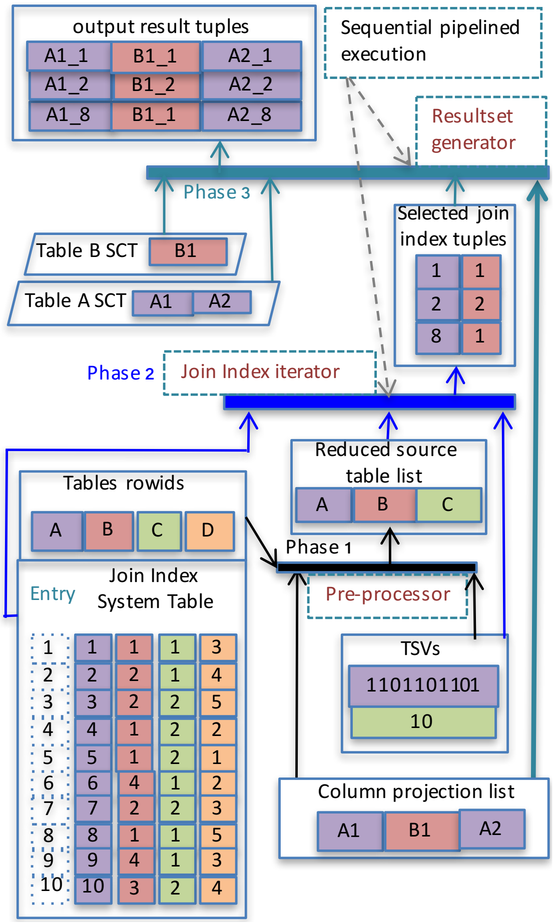
Implementation Details. As mentioned previously, the TSVs were originally designed as memory resident bit vectors, which were compressed and optimized for sequential access. However, as we will observe later, when we use the TSVs in connection with the join index, the lookup of bit positions in the TSV follows a random access. Our prototype testing of random access on the compressed list implementations of TSV proved this to be a potential bottleneck as lists used in the compressed format are not suited for random lookups. Thus, we switched to using uncompressed TSVs, which are stored as a contiguous 64-bit integer array, with each integer representing the status of 64 tuples in the partition. As in IDV the maximum number of records in a partition is 2 billion, the theoretical maximum size of an uncompressed TSV is 268 MB, which we consider acceptable given the performance gain of now being able to access a bit position in $\Theta(1)$. In case there are no predicate selections on a partition or all bits would be 1 (all tuples in the partition qualify), we do not maintain the TSV, as all tuples are selected. Instead, in the absence of a TSV, a TSV lookup is always set to return true.

3.2.2 *Join index query task*

The actual join execution is depicted in fig. 7 for a join over the four tables $A - D$ containing selection predicates for attribute $A1$ of table A and $C1$ of table C, that is, the previous execution step has created TSVs for tables A and C. Furthermore, the query projects on attributes $A1$ and $A2$ of table A and $B1$ of table B, i.e., $A1, A2$ and $B1$ need to be in the result set. We assume a join index that covers exactly the four tables $A - D$. Each partition of the join index is assigned to a different worker task that performs the join and the generation of the result set in three phases.

Phase 1. In phase 1, the worker task performs *pre-processing* steps aimed at reducing the I/O and CPU processing overhead by determining the set of tables and their attributes that are relevant for the query. The worker task goes through TSVs and the projection list specified in the query, and builds a reduced list of source tables. Only those tables are relevant that have a column in the projection list (tables A and B) or that have a TSV which results in tuple elimination (tables A and C). Thus, the join index columns corresponding to the rowids from tables A, B and C must be read by the worker task, in order to be able to check whether the bit for this rowid is set in the corresponding TSV, and if yes, retrieve the corresponding result set attributes. In contrast, there is no need for the worker task to read the join index column corresponding to the rowids from table D as it was not involved in the projection or selection. Thus, the reduced source table list consists of only A, B and C.

Phase 2. In this phase, the *join index iterator* determines the list of qualifying join index tuples. It will only read those columns from the join index system table that refer to the rowids of the tables appearing in the reduced source table list. For each of the join index tuples it checks whether the tuple qualifies for output which is the case if all selection predicates are fulfilled. More precisely, a join index tuple qualifies to generate output for the query, if all rowids in that join index tuple map to a 1 bit in the TSV of the corre-

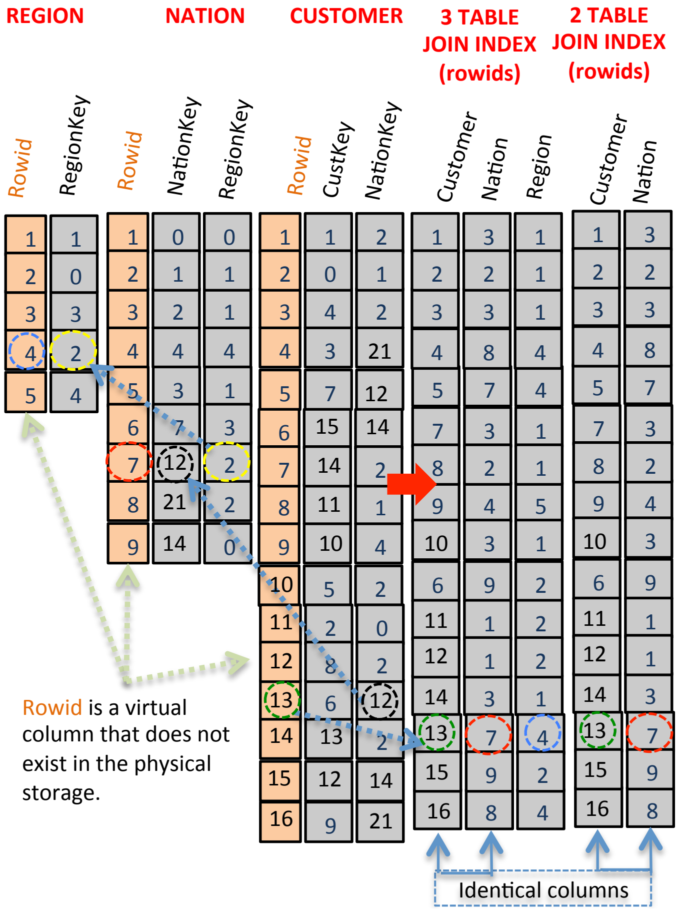


A1_1	B1_1	A2_1
A1_2	B1_2	A2_2
A1_8	B1_1	A2_8

1	1
2	2
8	1

Entry				
1	1	1	1	3
2	2	2	1	4
3	3	2	2	5
4	4	1	2	2
5	5	1	2	1
6	6	4	1	2
7	7	2	2	3
8	8	1	1	5
9	9	4	1	3
10	10	3	2	4

Figure 7: join index query task processing


REGION
NATION
CUSTOMER
3 TABLE JOIN INDEX (rowids)
2 TABLE JOIN INDEX (rowids)
Rowid is a virtual column that does not exist in the physical storage.
Identical columns

REGION

Rowid	RegionKey
1	1
2	0
3	3
4	2
5	4

NATION

Rowid	NationKey	RegionKey
1	0	0
2	1	1
3	2	1
4	4	4
5	3	1
6	7	3
7	12	2
8	21	2
9	14	0

CUSTOMER

Rowid	CustKey	NationKey
1	1	2
2	0	1
3	4	2
4	3	21
5	7	12
6	15	14
7	14	2
8	11	1
9	10	4
10	5	2
11	2	0
12	8	2
13	6	12
14	13	2
15	12	14
16	9	21

3 TABLE JOIN INDEX (rowids)

Customer	Nation	Region
1	3	1
2	2	1
3	3	1
4	8	4
5	7	4
7	3	1
8	2	1
9	4	5
10	3	1
6	9	2
11	1	2
12	1	2
14	3	1
13	7	4
15	9	2
16	8	4

2 TABLE JOIN INDEX (rowids)

Customer	Nation
1	3
2	2
3	3
4	8
5	7
7	3
8	2
9	4
10	3
6	9
11	1
12	1
14	3
13	7
15	9
16	8

Figure 8: Multi-table join indices and foreign key relationships

sponding source table partition. For tables without selection predicate, this bit lookup always returns 1 because all tuples of that table qualify.

Phase 3. A qualifying index tuples is then passed to phase 3, which comprises of the *resultset generator*. It uses rowids from the join index tuple to retrieve the attributes specified in the column projection list from the corresponding source table partitions to create the output result set.

Performance Discussion. The join index tuple selection (phase 2) and output resultset generation (phase 3) happen in a *sequential pipeline*, so that the process starts generating the output records before the join index is completely traversed. This helps significantly in reducing the first row generation time, as we do not wait for all the selected join index tuples to be processed before producing any output. Thus, the client does not have to wait for the completion of the query to start receiving the first results. This not only reduces the perceived response time but is also beneficial when only a sample of records is needed by the client.

The join index iterator reads the entire join index only once per query in a sequential fashion, ignoring columns of tables that were pruned in the pre-processing step.

Furthermore, only the source table blocks pertaining to the attributes required for the projection list or for evaluation of selection predicates will be read from disk. That is, our approach follows for a truly late materialization approach and is in tune with the principles of column-stores by avoiding I/O on irrelevant attributes. As any block fetched from disk is cached in memory, further lookups on the block do not incur a further physical I/O if the processing job is not memory bound.

3.3 One join index, many joins

Reducing the source table list to only relevant tables has some additional advantages. We can, under certain conditions, use an N-way join index over tables $T_1, T_2, ...T_n$ to evaluate queries that only join a subset of these tables. To understand this better, let us consider a simpler, single partition system of the 3-table join index described in fig. 5. The number of partitions are irrelevant for our discussion. The new join index arrangement is shown in fig. 8. Here we have a $1 : N$ mapping from REGION to NATION and $1 : M$ mapping from NATION to CUSTOMER. If *all* the foreign keys of the referencing relations in the join are *not nullable*, then we can make use of the original 3-table join index for a query that only joins CUSTOMER and NATION. This is because, if the foreign key column is not nullable, then the equi-join between the referencing relation and the referenced relation will always yield the same cardinality in the output relation as the original referencing relation. In the particular example, if every NATION tuple must refer to a REGION, then a join between NATION and REGION will have the same number of tuples as NATION. Hence the contents of a 2-table join index between CUSTOMER and NATION will be identical to the 3-table join index in fig. 5 without the rowid column for REGION, which in columnar storage can be ignored while reading.

Additionally, we can use an N-table join index over tables $T_1, T_2, ...T_n$ to evaluate queries that join a superset of these tables, i.e., joins that contain at least all tables covered by the N-table join index. In this case, the query can be processed by first using the join index and evaluating the N joins over $T_1, ...T_n$, persisting an intermediate relation T that contains the necessary attributes from these N relations and then joining T with the remaining relations using the conventional query processing steps.

4. SYSTEM EVALUATION

4.1 System configuration

The test environment databases was setup on a Dell XPS 9100 system having 8 Intel$^{\circledR}$ Core$^{\text{TM}}$i7 CPU 960 @ 3.20GHz processors with 4 cores, 12 GB 1333 MHz DDR3-SDRAM, 1 TB Western Digital WD10EALX 6 GB/s 7200RPM SATA storage, running Linux - Kubuntu 14.04 with 12 GB swap.

4.2 Benchmark

Our evaluation is based on the TPC-H benchmark as the industry-wide standard for decision support benchmarking. It has been widely used for benchmarking column stores like C-Store [21] and MonetDB [4]. Most of our experimental runs are performed against TPC-H databases of scale factor (SF) 1, 2, 4, 8, 12, 16, 20, 25 and 50. Our tests attempt to understand each of the different features of our design and how they work together. Thus, most tests consider only a single partition per table in order to focus on other dimensions. Whenever more than one partition per table is used, we state this explicitly.

We used modified versions of the TPC-H queries focusing on the selection predicates, the joins and the projection of simple attributes but without any aggregation components. We did not integrate this functionality into our prototype result set generator, as we are only interested in understanding the performance implications of the actual join and its interaction with selection and projection.

4.3 Experiments and Results

4.3.1 Two-table single partition joins

For this base experiment, we had only a single partition per table and we only consider the TPC-H queries Q12-14, Q16-17 and Q19, that are defined over two tables. We created a 2-table join index for each of these queries.

Additionally, the fact that each table has only a single partition also eliminates any redundant TSV evaluations in the original implementation. This a performance overhead that we discussed in section 3.2.1 with the original join query processing workflow. Therefore, with respect to the TSV evaluation strategy, this test case benefits the original implementation over the join index approach. This is because the original implementation performs the TSV evaluation and join in the same step, whereas our modified join workflow utilizing the join index performs the TSV evaluation first and persists the TSV. The join is performed only in the next step, introducing a small overhead for this test case for our join index based approach.

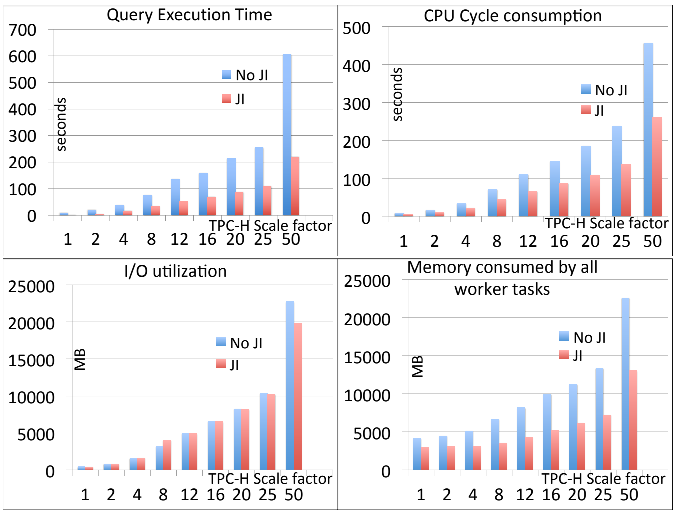


Figure 9: 2-table 1-partition join using queries Q12,13,14,16,17,19

Fig. 9 shows the total across all queries for execution time, CPU, I/O and memory consumption for executions with no join index (No JI) and with using the join index (JI) for the different scale factors of the database. Overall, by using the join indices, queries run about 60% faster than without join index. This is mainly due to the lower CPU costs of the join index based execution, as can be seen from the CPU utilization chart. Using the join index consumed only about 55% of the CPU compared with no index executions.

The I/O utilization, however, does not show any significant deviation when join indices are used by the query. This is because the worker tasks need to process an additional data structure that stores the join index system table, and any I/O savings that could be attributed to avoiding join computation is amortized over the cost of reading the join index. To understand this, we should consider that in the TPC-H database schema the key columns of the relations are of integer domain. While a query not using any index has to read the join attributes in order to compute the join, a query using a join index needs to read the rowids from the join index system table which are also of integer domain. Hence, intuitively, both kinds of queries have to execute approximately the same amount of I/O, in the absence of other influencing factors like projection attributes.

Analyzing the memory utilization, we notice about 45% reduction in the memory consumed by the worker tasks when using a join index. This can be attributed to the fact that in the absence of a join index, the worker tasks need to load the key columns into memory buffers to facilitate the merge join. With the join index based approach, our sequential scan technique requires only the current block (which is being processed) of the join index to be in the memory. Thus, the size of the join index does not have any significant memory impact. Also, sequential iteration of the join index is a CPU cache - friendly operation, a property that lends itself to faster program execution. Such *cache conscious* techniques of performance enhancements have been successfully employed in other column stores before [4].

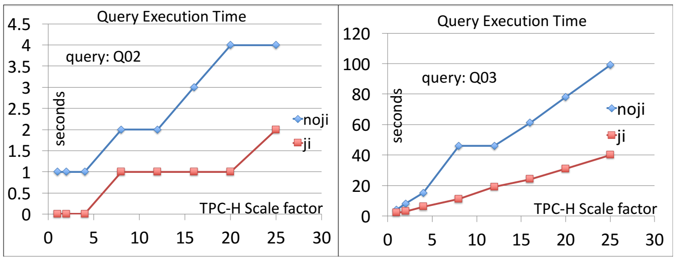


Figure 10: multi-table 1-partition joins

4.3.2 Multi-table single partition joins

Most of the real-world decision support queries are defined over many tables. Hence, for this test case we use two queries from the TPC-H benchmark suite. Q02 is a five table join between `Part`, `Supplier`, `Partsupp`, `Nation` and `Region`. Q03 is a three table join between `Lineitem`, `Orders` and `Customer`. To support these queries, we create the corresponding five-table and three-table join indices respectively. We use single partitions to isolate and observe the performance impact of having multiple tables in the join.

Fig. 10 shows the execution time for both queries again with increasing scale factor. Q03 runs 60% faster with the join-index based execution, similar to what we observed for two-table joins. Q02, however, behaves differently. The tables involved in Q02 are small in comparison to other tables like `Lineitem` or `Orders` which are involved in most of the other queries. The largest table involved in Q02 is `partsupp` at 20 million records for a scale factor 25 database. Also, the query itself only returns about 0.08% of the records, being very highly selective with its predicates. The selection predicate on the `part` table causes record elimination in the first join step with `partsupp`, resulting in reducing the size of intermediate tables in the succeeding join steps. Thus, this query is inherently fast in nature and, as can be observed by the execution time provided in the figure, takes only a few seconds even for large databases. This performance benefit of the join index based execution results from a combination of avoiding the join computation costs along with savings from late materialization. The later has a significant impact on this query's performance as it retrieves a significant number of attributes from different tables, making it an ideal candidate for savings from late materialization.

4.3.3 Two-table multi-partition joins

Multi-partitioned tables are the most common scenario in very large databases like the ones typically supported by IDV. For this test case, we setup 2, 3, 5 and 10 partition versions of the database having a scaling factor of 50, to facilitate the execution of the 2-table join query Q12. This query joins `Lineitem` and `Orders` tables which are the two largest tables in the TPC-H database. Further, the selection predicates on `Lineitem` limit the number of records retrieved by the query to 2.74% of `Lineitem` table. Having no join index, all partition combinations have to be joined, leading to the creation of $m \times n$ worker tasks, each performing the join of one partition combination.

In the case of using a join index, as discussed in section 3.1, due to the associative nature of data in partitions across related tables, we can often determine at index creation time, that a certain combination of partitions does not result in any joining tuples, and thus, in an empty join index partition. In fact, this was the case for our test database. For a 10 partition database, where both the tables involved in the join were partitioned to 10 equal size partitions, our index creation process materialized only 19 join index partitions instead of the theoretical maximum of $10 \times 10 = 100$ partitions. Thus, at the time of the join, only 19 worker tasks need to be spawned.

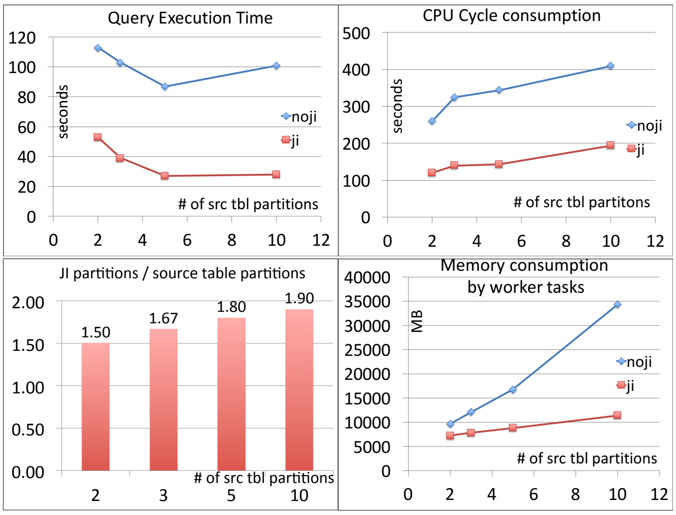


Figure 11: two-table multi-partition join query Q12

From fig. 11 we can see that the join index execution is always at least two times as fast as an execution without join index, and the performance gap increases with increasing number of partitions (top left figure).

At low number of partitions the performance benefit is due to the generally lower CPU and memory demands of the join index based execution as discussed in the previous experiments. When we now increase the number of partitions up to 5 partitions per table, performance improves for both strategies as the different partitions can be executed in parallel taking advantage of all cores and the ample available main memory. However, with 10 partitions, while the execution time of the join index implementation stagnates, performance becomes worse for the execution without join index. The reason for the latter are the much higher CPU and memory requirements (see the top and bottom right figures) as so many more partitions have to be read into main memory and joined even if they do not result in matching tuples. Once all compute cores and main memory have been used to exploit maximum parallelism a further increase in number of worker tasks due to the increase in partition combinations leads to too much contention and thus, performance decreases.

In contrast, the increase in CPU and memory overhead with increasing number of partitions is relatively small for the join index based execution. The reason is that the number of join index partitions per source table partition increases only slightly with the number of partitions (left bottom figure) and thus, the overall number of join index partitions remains relatively small. Our current system configuration can exploit the increased parallelism when increasing the number of partitions from 2 to 5, and still does not see any deterioration when there are 10 partitions.

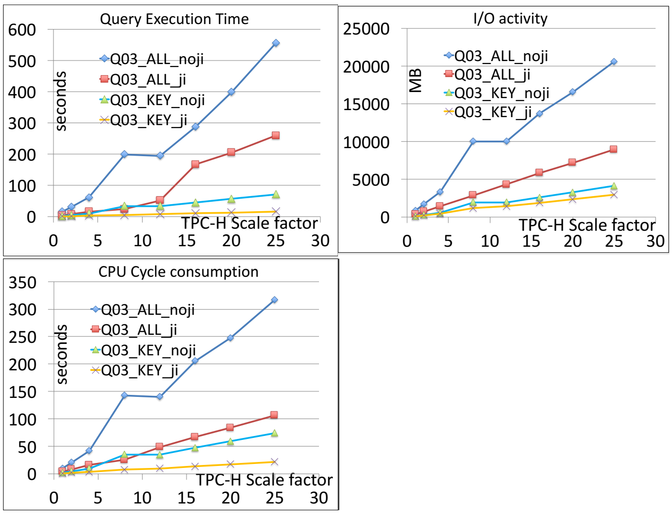


Figure 12: late materialization

4.3.4 *Late materialization*

To understand the impact of late materialization, we used the three table join query Q03 that joins `Lineitem`, `Orders` and `Customer`. For this test case, we created two versions of the query. The first version, Q03_ALL selects every attribute from `Orders` and `Customer` (the two relations joined first when no join index is used). The second version Q03_KEY selects only key attributes, reducing any impact due to late materialization. Thus, any increase in resource utilization from Q03_KEY to Q03_ALL will be the cost associated with materializing the extra attributes that are in the projection list of Q03_ALL. The queries were executed on single partition tables.

Fig. 12 shows execution times with increasing scale factor. Again, using a join index is always better than not using the index which is to be expected based on the previous test cases. Comparing the ALL version against the KEY version, we can observe that generating a result set with many attributes is generally expensive in both implementations. For the join index based execution, execution time for Q03_ALL compared to Q03_KEY for a database with scale factor 25 increases by 244 seconds, while it increases by 485 seconds when no index is used. Given that the attribute extraction proves to be a major part of the execution time, the benefit of late materialization becomes very apparent. The analysis performed on the difference in CPU consumption and I/O utilization correlate with our observation for query execution timings.

4.3.5 *Query selectivity*

To test the influence of query selectivity on performance, we took again the 2-table join query Q12 which joins `LineItem` and `Order` tables but changed its selection predicate on `LineItem` so that the percentage of records selected varied from 0.05% to 100%. Both the tables were composed of a single partition each. Fig. 13 shows the execution times for a scaling factor of 50. At 0.05%, the join-based implementation is 18% better than having no join index, but the gains quickly increase with larger selectivity percentages. That is, the benefits of using a join index increase with the number of records joined. The reason is that the join index execution needs to iterate over the entire index irrespective of the selection predicates, whereas the non-index based approach can reduce the number of records to be joined by applying the selection predicates in advance, reducing the CPU consumption for the join computation itself.

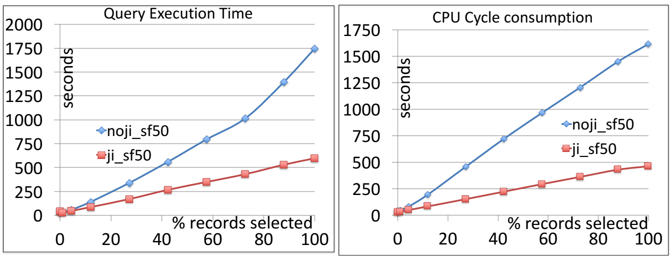


Figure 13: query Q12 at different selectivity

5. CONCLUSIONS

In this paper, we propose a join index implementation for a columnar archive store to facilitate faster query response times. Our implementation integrates seamlessly with the horizontally partitioned nature of the system which facilitates scalability and the columnar structure which allows for later materialization. N-join indices can be also exploited in an efficient manner for joins with less or more then N tables. Our performance evaluation using a TPC-H based benchmark over a variety of database and query characteristics demonstrate significant savings in execution time, and CPU and memory usage compared to an execution without join index.

We are presently exploring more efficient ways of storing rowids in the join index system table as well as runtime clustering of join index partitions at the query processing stage to increase main memory cache hit rates.

6. REFERENCES

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *ACM SIGMOD*, pages 967–980, 2008.

[2] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization Strategies in a Column-Oriented DBMS. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 466–475, 2007.

[3] J. A. Blakeley and N. L. Martin. Join Index, Materialized View, and Hybrid-Hash Join: a Performance Analysis. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 256–263, 1990.

[4] P. A. Boncz. *Monet; a Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, University of Amsterdam (UvA), May 2002.

[5] B. C. Desai. Performance of a Composite Attribute and Join Index. *IEEE Transactions on Software Engineering*, 15(2):142–152, 1989.

[6] Gartner. Worldwide Business Intelligence and Analytics Market 2016. *Published at http://www.gartner.com/newsroom/id/3198917*, 2016.

[7] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–169, 1993.

[8] R. Grondin, E. Fadeitchev, and V. Zarouba. Searchable Archive, Feb. 26 2013. US Patent 8,386,435.

[9] T. Haerder. Implementing a Generalized Access Path Structure for a Relational Database System. *ACM Transactions on Database Systems*, 3(3):285–298, 1978.

[10] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance Tradeoffs in Read-Optimized Databases. In *VLDB*, pages 487–498, 2006.

[11] Informatica Corporation. Informatica Data Archive Manage Application Data throughout its Lifecycle. https://www.informatica.com/content/dam/informatica-com/global/amer/us/collateral/data-sheet/data-archive_data-sheet_6955.pdf, Aug. 2014.

[12] M. Komorowski. A history of storage cost (update). *http://www.mkomo.com/cost-per-gigabyte-update*, 2014.

[13] Z. Li and K. A. Ross. Fast Joins Using Join Indices. *The VLDB Journal-The International Journal on Very Large Data Bases*, 8(1):1–24, 1999.

[14] S. Manegold, P. Boncz, N. Nes, and M. Kersten. Cache-Conscious Radix-Decluster Projections. In *VLDB*, pages 684–695, 2004.

[15] K. P. Mikkilineni and S. Y. W. Su. An evaluation of Relational Join Algorithms in a Pipelined Query Processing Environment. *IEEE Transactions on Software Engineering*, 14(6):838–848, 1988.

[16] P. Mishra and M. H. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

[17] P. O'Neil and G. Graefe. Multi-table Joins Through Bitmapped Join Indices. *ACM SIGMOD Record*, 24(3):8–11, 1995.

[18] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *ACM SIGMOD*, pages 165–178, 2009.

[19] H. A. Schmid and P. A. Bernstein. A Multi-Level Architecture for Relational Data Base Systems. In *VLDB*, pages 202–226, 1975.

[20] O. Shmueli and A. Itai. Maintenance of Views. In *ACM SIGMOD Record*, volume 14, pages 240–255, 1984.

[21] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-Store: a Column-Oriented DBMS. In *VLDB*, pages 553–564, 2005.

[22] Teradata. Teradata Columnar. *Published at http://www.teradata.com/teradata-columnar*, 2016.

[23] P. Valduriez. Join Indices. *ACM Transactions on Database Systems*, 12(2):218–246, 1987.

[24] Y. Zhang, S. Wang, and J. Lu. Improving Performance by Creating a Native Join-Index for OLAP. *Frontiers of Computer Science in China*, 5(2):236–249, 2011.

RDF Keyword-based Query Technology Meets a Real-World Dataset

Grettel M. García[1,2], Yenier T. Izquierdo[1,2], Elisa S. Menendez[1,2],
Frederic Dartayre[1], Marco A. Casanova[1,2]
[1]Instituto TecGraf – Pontifícia Universidade Católica do Rio de Janeiro
[2]Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 225 – Rio de Janeiro, RJ – Brazil CEP 22451-900
+55-21-3527-1500
ggarcia@inf.puc-rio.br, yizquierdo@inf.puc-rio.br, emenendez@inf.puc-rio.br,
fdartayre@tecgraf.puc-rio.br, casanova@inf.puc-rio.br

ABSTRACT

This paper presents the results of an industrial project, conducted by the TecGraf Institute and Petrobras (the Brazilian Petroleum Company), to develop a tool to facilitate access to a large database, with hydrocarbon exploration data, by combining RDF technology with keyword search. The tool features an algorithm to translate a keyword query into a SPARQL query such that each result of the SPARQL query is an answer for the keyword query. The algorithm explores the RDF schema of the RDF dataset to generate the SPARQL query and to avoid user intervention during the translation process. The tool offers an interface which allows the user to specify keywords, as well as filters and unit measures, and presents the results with the help of a table and a graph. Finally, the paper describes experiments which show that the tool achieves very good performance for the real-world industrial dataset and meets users' expectations. The tool was further validated against full versions of the IMDb and Mondial datasets.

CCS Concepts

Information systems → Information retrieval → Information retrieval query processing → Query reformulation

Keywords

Keyword search; SPARQL; RDF.

1. INTRODUCTION

Keyword search is typically associated with information retrieval systems, especially those designed for the Web. The user just specifies a few terms, called keywords, and it is up to the system to retrieve the documents, such as Web pages, that best match the list of keywords. These systems also usually offer an advanced search interface, which the user may take advantage to specify Boolean expressions involving the keywords, or limit the search to a subset of the documents, such as an Internet domain. Information retrieval systems typically implement algorithms to rank the results of a keyword search so that, hopefully, the user will find the most interesting documents at the top of the list. The success of such systems may therefore be credited to: (1) a very simple user interface; (2) an efficient document retrieval mechanism; and (3) a ranking algorithm which meets user expectations.

By contrast, database management systems offer sophisticated query languages to access structured data. It is up to the database applications to create user interfaces that hide the complexity of the query language. User interfaces are often designed as a stack of pages with numerous "boxes" that the user must fill with his search parameters. This traditional design may end up with uncomfortable user interfaces, which are amply justified, though, when the user has to specify exact data, such as a flight number and a flight date.

Hitting the middle ground, we find database applications that offer keyword search interfaces over conventional databases or, in short, *keyword search database applications*. These applications should reach a performance similar to information retrieval applications, despite the fact that the underlying data is stored in a conventional database. Furthermore, they should free the user from filling "boxes" with exact data by compiling keyword searches to the query language supported and by ranking the results in a meaningful way from the user point of view.

Keyword search applications over relational databases have been studied for quite some time. More recently, examples of such applications designed for RDF datasets have emerged. The adoption of RDF as the underlying data model has some interesting advantages. The most obvious is the flexibility RDF offers by modeling data as RDF triples of the form (s,p,o), which asserts that resource s has property p with value o. Of special interest for keyword search is the fact that RDF imposes no strict distinction between data and metadata, that is, a keyword may match the name or description of a class or of a property in the same way as it matches a data value. RDF management systems also sometimes offer an inference layer so that one may expand the stored RDF data with derived data in ways that surpass (relational) views. Thus, a keyword may match derived data as much as stored data. Lastly, an RDF dataset may be treated as a graph, which allows the use of graph concepts and algorithms to explore the data.

The paper summarizes the results of an industrial project, conducted by the TecGraf Institute and Petrobras (the Brazilian Petroleum Company), to facilitate access to a large relational database, with hydrocarbon exploration data, by combining RDF technology with keyword search. The prototype application is being deployed to the production environment to be tested on a large scaled by the target users.

The contributions of this paper are as follows. First, the paper defines the concept of an answer for a keyword-based query over an RDF dataset. Second, the paper introduces an algorithm to translate keyword-based queries to SPARQL queries that takes advantage of the schema of the RDF dataset to avoid user intervention and achieve good performance, even for large RDF datasets. Third, it describes an interface that allows the user to specify keywords with the help of an auto-completion feature, as well as filters and unit measures, such as "wells with depth between 1,000m and 2,000m". The interface presents an answer to the user in a way that reduces the cognitive overhead required to navigate through an RDF graph. Lastly, the paper describes experiments that show that the tool achieves very good performance for the real-world industrial dataset and meets users' expectations. The tool was further validated against full versions of the IMDb and Mondial datasets.

The remainder of this paper is organized as follows. Section 2 summarizes related work. Section 3 provides a brief background on RDF and defines the basic concepts of keyword-based queries. Section 4 covers the translation algorithm and the user interface. Section 5 describes experiments to assess the performance and usability of the tool. Finally, Section 6 contains the conclusions.

2. Related Work

Keyword-based query processing. Tools that implement keyword-based queries over relational databases and RDF datasets have been investigated for some time. We may distinguish between tools that are *schema-based*, in the sense that they use information about the conceptual schema to compile a keyword-based query into an SQL or SPARQL query, from those that are *graph-based*, in the sense that operate directly on the data. We may also identify *pattern-based* tools, which hit the middle ground, in the sense that they mine patterns from the RDF dataset to be used in lieu of the conceptual schema. It is also useful to distinguish between *fully automatic* tools from tools that resort to user intervention during the processing of the keyword-based queries.

BANKS [1] and BLINKS [11] are examples of early relational graph-based tools. Relational schema-based tools explore the foreign keys declared in the relational schema to compile a keyword-based query into an SQL query with a minimal set of join clauses, based on the notion of *candidate networks* (CNs). This approach was first proposed in DISCOVER [12] and DBXplorer [2] and adopted in a quite a few tools, including recent ones [15].

SPARK [28] offers an example of an early pattern-based RDF graph-based tool. Tran et al. [21] combine the idea of generating summary graphs for the original RDF graph, using the class hierarchy, to generate and rank candidate SPARQL queries. Zhang et al. [26] investigated a solution to this problem, backed up by experiments over a subset of the original IMDb, a selection of articles from Wikipedia, and the Mondial dataset. More recently, Yang et al. [24] proposed to mine tree patterns that will then connect together the keywords specified by the user; the tree patterns are ordered by relevance using their size, the pagerank of the nodes and the quality of keyword match. Zheng et al. [27] proposed a systematic method to mine semantically equivalent structure patterns to summarize the knowledge graph and, thereby, circumvent the lack of an RDF schema. Finally, De Virgilio [7] proposed an RDF keyword-based query processing strategy based on tensor calculus, later extended to a distributed environment [8].

QUICK (*QUery Intent Constructor for Keywords*) [25] is an RDF schema-based tool designed to translate keyword-based queries to SPARQL queries with the help of the users, who choose a set of intermediate queries, that the tool ranks and executes.

The tool described in this paper is schema-based and fully automatic. We borrowed from the early relational graph-based tools the idea of minimizing the number of equijoins by generating a Steiner tree of a graph induced by the RDF schema. However, we introduce the (new) concept of a *nucleus*, consisting of a class, a list of properties, and a list of property values, which is in some sense analogous to a tuple and helps translate keyword-based queries to SPARQL queries. The Steiner tree will then connect the classes of the nucleus that cover the keywords.

QUICK is the tool closest to ours in so far as both tools explore the RDF schema to synthesize SPARQL queries. However, differently from QUICK, we opted for a fully automatic translation. This was possible essentially because our tool was designed to operate over an RDF dataset which has a rich schema and whose data exhibits low ambiguity.

Triplification of the relational database. Triplification, the process of mapping a relational database to an RDF dataset, is based on well-established technologies, backed up by a standardized mapping language, R2RML [6]. However, relational databases are usually normalized and, therefore, should not be directly mapped to RDF. To deal with this issue, we followed the strategy proposed in [22], which suggests to first create relational views that define an unnormalized relational schema and then write the R2RML mappings on top of these views.

In fact, the judicious design of the RDF schema helps the translation process from keyword-based queries to SPARQL queries. This requires additional comments. First, the assumption that the RDF dataset has a known schema should not be viewed as a demerit. Indeed, a large fraction of the LOD datasets do have a known schema (vocabulary or ontology) [17]. Furthermore, in a corporate environment, such as ours, RDF datasets are frequently triplifications of relational databases. Second, even when one cannot change the (relational or RDF) schema, one may add a conceptual layer, defined with the help of views, that hide normalizations, in the relational case, or poorly designed RDF schemas, which in both cases would lead to ambiguities when processing keyword-based queries.

Benchmarks. Coffman and Weaver [4] describe a benchmark which uses a simplified, relational version of IMDb, a subset of Wikipedia, and a subset of the Mondial dataset. The keyword queries are mostly very simple.

Guo et al. [9] introduced LUBM, a benchmark for OWL knowledge base systems, which consists of an ontology for the university domain, synthetic OWL data scalable to an arbitrary size, and 14 SPARQL queries. More recently, an ontology-based data access benchmark, the *NPD Benchmark* [13][20], was constructed using real data from the Norwegian Petroleum Directorate (NPD) FactPages. The benchmark generates, from the NPD data, datasets of increasing size; the SPARQL queries were formulated by domain experts from an informal set of questions provided by regular users of the FactPages. Finally, Qiao and Özsoyoğlu [18] published the RBench, an application-specific RDF benchmarking tool that takes an RDF dataset from any application as a template, and generates a set of synthetic datasets and different types of queries systematically.

Although our tool was designed for a specific RDF dataset, we decided to test it against other datasets. However, a direct comparison with other keyword search tools turned out to be problematic, for two basic reasons. First, contrasting with Coffman's benchmark setting, our tool takes advantage of more complex RDF schemas and of keyword-based queries with a fairly large number of keywords – the query profile of our typical users – to avoid user intervention during the synthesis of the SPARQL query. Second, our tool presently does not incorporate reasoning features, i.e., we deal with a standard dataset and not with a knowledge base. In the end, we opted to further test our tool against the full versions of IMDb and Mondial, which feature conceptual schemas with a complexity closer to the schema of the target industrial dataset. We used the same list of keyword queries as in Coffman's benchmark, albeit they are much simpler than those expected from our typical users, as already pointed out.

3. BASIC DEFINITIONS

3.1 RDF Essentials

In this section, we summarize some basic concepts pertaining to the *Resource Description Framework* (RDF) [5]. The reader familiar with RDF may skip this section.

An *Internationalized Resource Identifier* (*IRI*) is a global identifier that denotes a resource. We will use the terms IRI and resource interchangeably. A *literal* is a basic value, such a string, a number, or a date. A *blank node* acts as a local identifier; a blank node can always be replaced by a new, globally unique IRI (a *Skolem IRI*). An *RDF term* is either an IRI, a blank node or a literal. The sets of IRIs, blank nodes and literals are disjoint. In the rest of this paper, ***IRI*** denotes the set of all IRIs and $\boldsymbol{L}$ the set of all literals.

RDF models data as triples of the form (s,p,o), where s is the *subject*, p is the *predicate* and o is the *object* of the triple. An RDF triple (s,p,o) says that some relationship, indicated by p, holds between the subject s and object o. The subject of a triple is an IRI or a blank node, the predicate is an IRI, and the object is an IRI, a blank node or a literal.

A set T of RDF triples, or an *RDF dataset*, is equivalent to a labeled graph G_T such that the set of nodes of G_T is the set of RDF terms that occur as subject or object of the triples in T and there is an edge (s,o) in G_T labeled with p iff the triple (s,p,o) occurs in T. Therefore, we will use the concepts of RDF dataset and RDF dataset graph interchangeably. Note that an IRI may occur both as a node and as an edge label in the same graph.

RDF offers enormous flexibility but, apart from the rdf:type property, which has a predefined semantics, it provides no means for defining application-specific classes and properties. Instead, such classes and properties, and hierarchies thereof, are described using extensions to RDF provided by the *RDF Schema 1.1* (RDF Schema or RDF-S) [3]. In RDF-S, a *class* is any resource having an rdf:type property whose value is the qualified name rdfs:Class of the RDF Schema vocabulary. Likewise, a *property* is any resource having an rdf:type property whose value is the qualified name rdfs:Property. The rdfs:domain property is used to indicate that a given property applies to a designated class, and the rdfs:range property is used to indicate that the values of a particular property are instances of a designated class or, alternatively, are instances (i.e., literals) of an XML Schema datatype. RDF-S also offers the rdfs:subClassOf and the rdfs:subPropertyOf properties that allow the specification of sub-class and sub-property axioms. Finally, RDF-S features a property, rdfs:comment, used to associate a comment with a resource, and a property, rdfs:label, used to assign a name to a resource.

An *RDF schema* is a set S of RDF triples that use the RDF-S vocabulary to declare classes, properties, property domains and ranges, and sub-class and sub-property axioms. Viewed as a set of RDF triples, S is also equivalent to a labelled graph G_S.

A *simple RDF schema* is a RDF schema that contains only class declarations, object and datatype property declarations and sub-class axioms (and no sub-property axioms). We then introduce a labelled graph, D_S, called an *RDF schema diagram*, defined as follows: (1) the nodes of D_S are the classes declared in S; and (2) there is an edge from class c to class d labelled with *subClassOf* iff c is declared as a subclass of d in S, and there is an edge from class c to class d labelled with p iff p is declared in S as an object property with domain c and range d.

Very briefly, we say that an RDF dataset T *follows* an RDF schema S iff we have: (1) $S \subseteq T$; (2) all classes and properties used in T, except those in S itself, are declared in S; and (3) the triples in T, again except those in S, satisfy all restrictions imposed by the declarations in S [3]. Note that, by this definition, the RDF schema is contained in the RDF dataset, which is convenient for our purposes (see Section 3.2).

Finally, SPARQL is a query language specifically designed to access RDF datasets [10]. SPARQL offers two types of queries. A *SELECT query* returns tabular data, whereas a *CONSTRUCT query* returns a set of RDF triples. The body of a SPARQL query is a list of *triple patterns*, defined like RDF triples, except that the subject, predicate or object can be a variable. The evaluation of a SPARQL query Q against an RDF dataset T binds values to the variables using a *solution mapping* σ in such a way that the WHERE clause of Q generates a subgraph of T. An *answer* of Q is an instantiation of the variables in the target clause of Q generated by σ.

3.2 Keyword-Based Queries

Let T be an RDF dataset and G_T be the corresponding RDF graph. We assume that T follows an RDF schema S, with $S \subseteq T$.

A *keyword-based query* K is simply a set of literals, or *keywords*.

Recall that $\boldsymbol{L}$ is the set of all literals. Let $match$: $\boldsymbol{L}\times\boldsymbol{L} \rightarrow [0,1]$ be a similarity function between literals such that $match(s,t)=j$ indicates how similar s and t are: j=1 says that s and t are identical, and j=0 indicates that s and t are completely dissimilar. We also introduce a *similarity threshold* $\sigma\in(0,1]$. We leave *match* and σ unspecified at this point.

The set $\boldsymbol{MM[K,S]}$ of *metadata matches* between K and the metadata descriptions of the classes and properties in S is defined as:

$$\boldsymbol{MM[K,T]} = \{\ (k,(r,p,v))\in K\times T\ /\ (r,p,v)\in S \wedge match(k,v)\geq\sigma\ \}$$

The set $\boldsymbol{VM[K,T]}$ of *property value matches* between K and property values of T is defined as (recall that $S \subseteq T$):

$$\boldsymbol{VM[K,T]} = \{\ (k,(r,p,v))\in K\times T\ /\ (r,p,v)\notin S \wedge match(k,v)\geq\sigma\ \}$$

The set of *matches* between K and T is then defined as:

$$\boldsymbol{M[K,T]} = \boldsymbol{MM[K,T]} \cup \boldsymbol{VM[K,T]}$$

An *answer* for K over T is a subset A of T such that:

(1) There is a subset of K, denoted K/A, such that, for each $k\in K/A$:
   a. There are (s,rdf:type,c_n), (c_n,rdfs:subClassOf,c_{n-1}),..., (c_1,rdfs:subClassOf,c_0) and (c_0,p_0,v_0) in A such that $(k,(c_0,p_0,v_0))\in\boldsymbol{MM[K,T]}$; or
   b. There are (s,q_n,v_n), (q_n,rdfs:subPropertyOf,q_{n-1}),..., (q_1,rdfs:subPropertyOf,q_0) and (q_0,p_0,v_0) in A such that $(k,(q_0,p_0,v_0))\in\boldsymbol{MM[K,T]}$; or
   c. There is $(r,p,v)\in A$ such that $(k,(r,p,v))\in\boldsymbol{VM[K,T]}$.

(2) There is no other answer B for K over T such that $K/A \subset K/B$.

We say that K/A is the set of keywords *matched* by A.

Condition (1a) says that a keyword k has a class metadata match for a class c_0 and the answer A must contain an instance of c_0 or one of its sub-classes c_n, in which case A must include all triples indicating that c_n is a sub-class of c_0. Likewise, Condition (1b) says that a keyword k has a property metadata match for a property q_0 and the answer A must contain an instance of q_0 or one of its sub-properties

q_n, in which case A must include all triples indicating that q_n is a sub-property of q_0. Condition (1c) simply says that k matches the literal of a triple (r,p,v) in A. Also, Condition (1) does not require that all keywords in K be matched in an answer. Indeed, we say that A is *total* iff $K/A = K$, and *partial* otherwise. Condition (2) requires that an answer must match as many keywords in K as possible.

The definition of an answer is quite liberal. In particular, it allows an answer A to be a set of disconnected triples, as in Figure 1c. To circumvent this problem, we define a partial order between answers as follows. Given a directed graph G, let $|G|$ denote the number of nodes and edges of G and $\#c(G)$ denote the number of connected components of G, when the direction of the edges of G is disregarded. We define a partial order "<" for graphs such that, given two graphs G and G',

$$\begin{aligned} G < G' \text{ iff } &(\#c(G) + |G|) < (\#c(G') + |G'|) \text{ or} \\ &(\#c(G) + |G|) = (\#c(G') + |G'|) \text{ and } \#c(G) < \#c(G') \end{aligned}$$

We use the partial order "<" between graphs to compare answers. We say that an answer A *is smaller than* an answer B iff $G_A < G_B$, where G_A and G_B are the RDF graphs of A and B (which may include metadata, since the RDF schema is part of the dataset). An answer A for K over T is *minimal* iff there is no other answer B for K over T such that $G_A < G_B$.

In this paper, we focus on heuristics to find, possibly, minimal answers for keyword-based queries. We are especially interested in heuristics that, given a keyword-based query K, generate a CONSTRUCT SPARQL query Q over T which is a *correct query interpretation* for K, in the sense that each set of triples returned by Q is an answer for K over T and, preferably, a minimal answer.

Example 1: Consider an RDF dataset T, whose RDF graph G_T is shown in Figure 1a, where the darker boxes with boldface italic labels partly denote the RDF schema. Consider the keyword-based query K = {`Mature`, `Sergipe`}. Then, we have the following set of matches of K for T:

$$\begin{aligned} \boldsymbol{M[K,T]} = \{ &(\texttt{Mature}, (r_1, \text{:stage}, \text{“Mature”})), \\ &(\texttt{Mature}, (r_2, \text{:stage}, \text{“Mature”})), \\ &(\texttt{Sergipe}, (r_1, \text{:inState}, \text{“Sergipe”})), \\ &(\texttt{Sergipe}, (r_3, \text{:name}, \text{“Sergipe Field”})) \} \end{aligned}$$

There are several possible answers for K over T, two of which are represented in Figures 1b and 1c (in the form of their RDF graphs; note that the dashed node labelled "Alagoas" is not part of answer A_2; it is depicted just to alert that A_2 includes resource r_2). Note that answers A_1 and A_2 match both keywords in K. However, since $|G_{A1}|=5$, $|G_{A2}|=6$, $\#c(G_{A1})=1$, and $\#c(G_{A2})=2$, we have $G_{A1} < G_{A2}$, and hence A_1 should be preferred to A_2.

However, the keyword-based query K is ambiguous, since it does not indicate whether the keyword `Sergipe` refers to a state or to an oil field. To disambiguate, we might consider the keyword-based query K' = { `Mature`, `"located in"`, `"Sergipe Field"`}. Indeed, we would obtain answer A_3, shown in Figure 1d. The dashed rectangle highlights components of the RDF schema which are part of the answer. Indeed, note that there is a property metadata match between the keyword "`located in`" and the label value "located in" of property :locIn (note again that the dashed node labelled "Alagoas" is not part of the answer). Furthermore, as required by the definition of an answer, note that $(r_2, \text{:locIn}, r_3)$ is an instance of property :locIn. Naturally, a second answer to K', similarly defined but involving resource r_1, would also be acceptable, since r_1 represents a mature well and is located in the Sergipe Field.

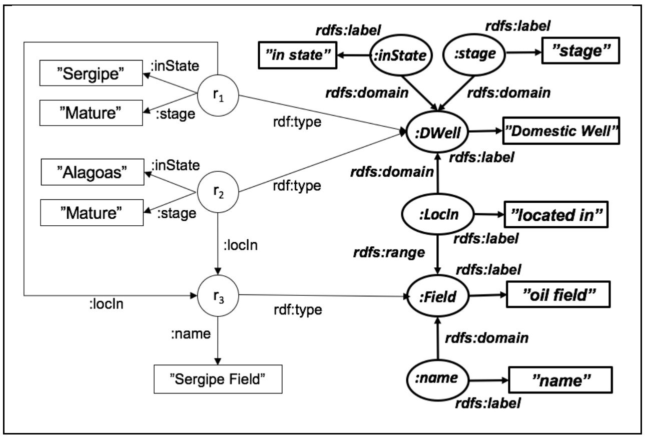


Figure 1a. The RDF graph *G* of Example 1.

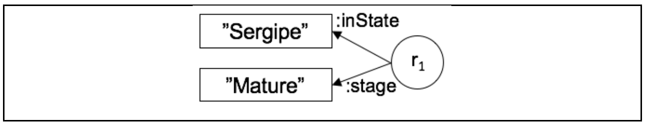


Figure 1b. An answer A_1 for the keyword-based query K = {`Mature`, `Sergipe`}.

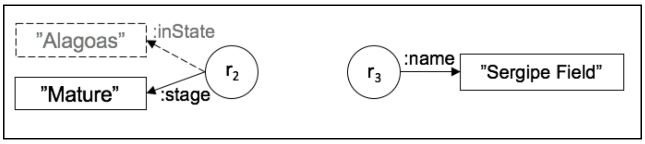


Figure 1c. A second answer A_2 for the keyword-based query K = {`Mature`, `Sergipe`}.

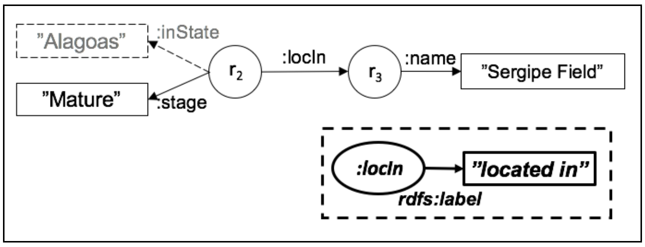


Figure 1d. An answer A_3 for the keyword-based query K' = {`Location`, `Dakota`, `Actor`, `Washington`}.

4. TRANSLATION OF KEYWORD QUERIES TO SPARQL QUERIES

4.1 Overview of the Translation Algorithm

The translation algorithm accepts a keyword-based query K and an RDF dataset T, and outputs a SPARQL query Q, which is a correct interpretation for K, in the sense that any result of Q is an answer for K over T. It assumes that T follows a simple RDF schema S.

In what follows, let G_T denote the RDF graph corresponding to T and D_S denote the RDF schema diagram of S.

Given a set of metadata matches $\boldsymbol{MM[K,T]}$ and a set of property value matches $\boldsymbol{VM[K,T]}$, we define two functions that group all keywords that match the same class or property:

$\boldsymbol{mm}[K,T] : \boldsymbol{IRI} \rightarrow 2^K$ such that
$\boldsymbol{mm}[K,T](r)=\{k{\in}K \,/\, (\exists p{\in}\boldsymbol{IRI})(\exists v{\in}\boldsymbol{L})((k,(r,p,v)){\in}\boldsymbol{MM[K,T]})\}$

$\boldsymbol{vm}[K,T] : \boldsymbol{IRI} \rightarrow 2^K$ such that
$\boldsymbol{vm}[K,T](q)=\{k{\in}K \,/\, (\exists s{\in}\boldsymbol{IRI})(\exists v{\in}\boldsymbol{L})((k,(s,q,v)){\in}\boldsymbol{VM[K,T]})\}$

We then define a *nucleus* as a triple N=(C,PL,PVL), where

(1) $C=(K_0,c)$, with K_0 = $\boldsymbol{mm}[K,T](c)$, such that c is a class of S

(2) PL={$(K_1,p_1),...,(K_m,p_m)$}, with K_i = $\boldsymbol{mm}[K,T](p_i)$, such that c is the domain of a property p_i of S, for $i\in[1,m]$

(3) PVL={$(K_{m+1},q_1),...,(K_{m+n},q_n)$}, with K_j = $\boldsymbol{vm}[K,T](q_j)$, such that c is the domain of a property q_j of S, for $j\in[1,n]$

Note that, since there might be several keywords that match the same element of N, we consider sets of keywords, rather than a single keyword. Furthermore, since a keyword may match more than one element of N, we do not require that K_i and K_j be disjoint, for $0 \le i \ne j \le m+n$. We say that N *covers* the set of keywords $K_N = K_0 \cup K_1 \cup ... \cup K_m \cup K_{m+1} \cup ... \cup K_{m+n}$.

Given a set of nucleuses $\boldsymbol{N}$={$N_1,\ldots,N_m$}, we also say that $\boldsymbol{N}$ *covers* $K_{N1} \cup \ldots \cup K_{Nm}$. Furthermore, we denote by $\boldsymbol{N_C}$ the set of classes of the nucleuses in $\boldsymbol{N}$ (which are nodes of D_S).

The translation algorithm implements two heuristics, called the *scoring* and the *minimization* heuristics. Intuitively, the scoring heuristic tries to capture the user intensions expressed in the list of keywords of a keyword-based query. Briefly, the scoring heuristic: (1) considers how good a match is, say "city" matches "Cities" better than "Sin City"; (2) assigns a higher score to metadata matches, on the grounds that, if the user specifies a keyword, say "city", that matches both a class label, say, "Cities", and the property value of an instance, say the film title "Sin City", then the user is probably more interested in the class labelled "Cities" than the specific film "Sin City"; (3) assigns a higher score to nucleuses that cover a larger number of keywords. The heuristic is formalized by defining a *score function* for the nucleuses, as follows.

Given a nucleus N=(C,PL,PVL), the *score* of N, denoted *score*(N), is the summation of all matches that N expresses, weighted by the type of the matches:

$$score(N) = (\alpha s_C + \beta s_P + (1-\alpha-\beta) s_V)$$

with

$$s_C = meta\_sim((K_0, c))$$

$$s_P = \sum\nolimits_{(K_i,p_i)\in PL} meta\_sim((K_i,p_i))$$

$$s_V = \sum\nolimits_{(K_j,q_j)\in PVL} value\_sim((K_j,q_j))$$

where

- α and β, with $0 < \alpha + \beta \le 1$, are parameters that weight between s_C, s_P and s_V, and which are experimentally set
- s_C is the combined score of the metadata matches for class c
- s_P is the combined score of the metadata matches for the properties in PL
- s_V is the combined score of the property value matches in PVL
- *meta_sim*((K,c)) is the sum of metadata match scores of class c
- *meta_sim*((K,p_i)) is the sum of metadata match scores of property p_i in PV
- *value_sim*((K,q_j)) is the sum of property value match scores of property q_j in PVL

Section 4.2 illustrates how to estimate *meta_sim* and *value_sim* and compute the score of a nucleus.

The minimization heuristic tries to generate minimal answers, in two stages. Ideally, we should try to find the smallest set of nucleuses that covers the largest set of keywords and that has the largest combined score. However, this is an NP-complete problem (by a reduction to the bin packing problem). The first stage of the minimization heuristic then implements a greedy algorithm that prioritizes the nucleuses with the largest scores and generates a set $\boldsymbol{N}$ of nucleuses such that:

(1) $\boldsymbol{N}$ covers a large subset of K.

(2) All nodes in $\boldsymbol{N_C}$ are in the same connected component of D_S.

where, we recall, $\boldsymbol{N_C}$ denotes the set of classes of the nucleuses in $\boldsymbol{N}$ (which are again nodes of the RDF schema diagram D_S).

If we synthesized a SPARQL query Q based only on the nucleuses in $\boldsymbol{N}$, then an answer of Q – which would induce an answer for the keyword-based query K – would have as many connected components as there are classes in $\boldsymbol{N_C}$. Since answers are measured in terms of the number of nodes and connected components, this situation would be unsatisfactory. The second stage of the minimization heuristic then forces an answer to have a single connected component by connecting the classes in $\boldsymbol{C_N}$, using a small number of edges of D_S. This is equivalent to generating a Steiner tree ST of D_S whose nodes are the classes in $\boldsymbol{N_C}$. Then, the algorithm uses the edges of ST to generate equijoin clauses of the SPARQL query Q in such a way that any answer of Q indeed has a single connected component.

Note that ST exists since all nodes in $\boldsymbol{N_C}$ belong to the same connected component of D_S, by (2). Furthermore, note that we use the RDF schema diagram D_S, and not the RDF dataset graph G_T. In fact, this is the only step of the algorithm that depends on the assumption that T has a schema S.

Figure 2 shows a high level description of the translation algorithm, while Section 4.2 illustrates the synthesis of a SPARQL query.

Step 1 removes stop words from K and matches the remaining elements in K with literals in T, creating a set of metadata matches $\boldsymbol{MM}[K,T]$ and a set of property value matches $\boldsymbol{VM}[K,T]$, as defined in Section 3.2. Step 1 uses auxiliary tables to speed up computing matches (see also Section 4.2). For each class declared in S, the ClassTable table stores the IRI, label, description and other property values declared in S for the class. The PropertyTable stores the property metadata, as for the classes. The JoinTable stores domains and ranges declared in S. A forth table, ValueTable, stores all distinct property value pairs that occur in T.

Step 2 uses $\boldsymbol{MM}[K,T]$ and $\boldsymbol{VM}[K,T]$ to compute a set $\boldsymbol{M}$ of nucleuses as follows. It first processes class metadata matches, generating *primary nucleuses*; all class metadata matches with the same class will be mapped to a single nucleus. Then, it processes property metadata matches, creating the property lists of the primary nucleuses, or generating *secondary nucleuses*, for properties whose domains are not in any primary nucleus; finally, it processes property value matches, creating the property value lists of the existing nucleuses, or generating new *secondary nucleuses*, again for those properties whose domains are not in any previously constructed nucleus.

Step 3 computes the score of each nucleus in $\boldsymbol{M}$, as defined above.

Step 4 corresponds to the first stage of the minimization heuristic and creates a set $\boldsymbol{N}$ of nucleuses as follows. It first adds to $\boldsymbol{N}$ the nucleus N_0 in $\boldsymbol{M}$ with the largest score, removing it from $\boldsymbol{M}$. Let H_0 be the connected component of the RDF schema diagram D_S that contains the class of N_0. It also removes from $\boldsymbol{M}$ all nucleuses whose classes are not in H_0. This guarantees that Step 5 will be able to run correctly. Let K_{N0} be the set of keywords covered by N_0. The keywords in K_{N0} need no longer be considered and are disregarded from the nucleuses remaining in $\boldsymbol{M}$, which therefore have their scores recomputed. Step 4 continues by adding to $\boldsymbol{N}$ the nucleus in $\boldsymbol{M}$ with the largest (recomputed) score that covers a keyword not covered by any of the nucleuses previously selected. Since such

```
Translation Algorithm:
Input:   A keyword query $K$
         An RDF dataset $T$, with a simple RDF schema $S$
Output:  A SPARQL query $Q$ over $T$
1.   Keyword matching:
     1.1.  Eliminate stop words from $K$.
     1.2.  Match each keyword with the classes, properties and property values in $G_T$, returning the set of metadata matches
           $MM[K,T]$ and the set of property value matches $VM[K,T]$, as defined in Section 3.2.2.
2.   Nucleus generation:
     2.1.  $M$ = empty.
     2.2.  For each class $c$ such that there is a class metadata match for $c$ in $MM[K,T]$, do:
           2.2.1.  If a nucleus with class $c$ does not exist in $M$,                        /* a metadata match for $d$ exists  */
                   add to $M$ a primary nucleus $N = ((K_c,c),\varnothing,\varnothing)$, with $K_c = mm[K,T](c)$.   /* which implies $K_c \neq \varnothing$  */
     2.3.  For each property $p$ such that there is a property metadata match for $p$ in $MM[K,T]$, do:
           2.3.1.  Let $d$ be the domain of $p$. If a nucleus $N$ with class $d$ does not exist in $M$, /* no primary nucleus exists for d */
                   add to $M$ a secondary nucleus $N = ((K_d,d),\varnothing,\varnothing)$, with $K_d = \varnothing$.   /* which implies $K_d = \varnothing$  */
           2.3.2.  Let $N$ be the nucleus with class $d$. Add $(K_p,p)$, with $K_p = mm[K,T](p)$, to the property list of $N$.
     2.4.  For each property $q$ such that there is a property value match for $q$ in $VM[K,T]$, do:
           2.4.1.  Let $d$ be the domain of $q$. If a nucleus $N$ with class $d$ does not exist in $M$, /* no nucleus exists for d  */
                   add to $M$ a secondary nucleus $N= ((K_d,d),\varnothing,\varnothing)$, with $K_d = \varnothing$.   /* which implies $K_d = \varnothing$  */
           2.4.2.  Let $N$ be the nucleus with class $d$. Add $(K_q,q)$, with $K_q = vm[K,T](q)$, to the property value list of $N$.
3.   Nucleus score computation:
       3.1. Compute the score of each nucleus in $M$.
4.   Nucleus selection:
     4.1.  Initialize a set $N$ with the nucleus $N_0$ in $M$ with the largest score and remove $N_0$ from $M$.
     4.2.  Let $D_S$ be the RDF schema diagram of $S$ and $H_0$ be the connected component of $S$ that contains the class of $N_0$.
           Remove from $M$ all nucleuses whose classes are not in $H_0$.
     4.3.  Update the sets of keywords and scores of the remaining nucleuses in $M$ by dropping the keywords covered by $N_0$.
     4.4.  While  there are keywords not covered by the nucleuses in $N$ and
                  there is a nucleus in $M$ that covers such keywords do:
           4.4.1.  Add to $N$ the nucleus $N_s$ in $M$ with the largest score such that $N_s$ covers such keywords.
           4.4.2.  Remove $N_s$ from $M$.
           4.4.3.  Update the sets of keywords and scores of the remaining nucleuses in $M$ by dropping the keywords covered
                   by $N_s$.
5.   Steiner tree generation:
     5.1.  Let $D_S$ again be the RDF schema diagram of $S$.
           Compute a Steiner tree $ST$ of $D_S$ that contains the set of classes of the nucleuses in $N$.
6.   Synthesis of the SPARQL query $Q$:
     6.1.  Construct the WHERE and the TARGET clauses of $Q$ from the nodes and edges of $ST$ and the nucleuses in $N$.
     6.2.  Return $Q$.
```

Figure 2. Outline of the Translation Algorithm.

nucleuses are selected from $\boldsymbol{M}$, they necessarily have a class that is in H_0. It stops when all keywords in K are covered by the nucleuses in $\boldsymbol{N}$, or when no nucleus in $\boldsymbol{M}$ covers an uncovered keyword.

Step 5 implements the second stage of the minimization heuristic. It computes an (approximated) minimal Steiner tree ST of the RDF schema diagram D_S that covers $\boldsymbol{N_C}$, the set of nodes that correspond to the classes of the nucleuses in $\boldsymbol{N}$.

Although not shown in Figure 2, Step 5 proceeds as follows. It first computes a new labelled directed graph G_N whose nodes are those in $\boldsymbol{N_C}$ and there is an edge (m,n) in G_N labelled with k iff the shortest path in the RDF schema diagram D_S connecting nodes m and n has length k. Then, Step 5 computes a minimal directed spanning tree TN for G_N. If no such directed spanning tree exists, then Step 5 tries to compute a minimal spanning tree TN for G_N, but ignoring the edge direction. TN will then induce the desired Steiner tree ST of D_S covering the nodes in N_C by simply replacing each edge of TN by the corresponding path in D_S.

Step 6 synthesizes a CONSTRUCT query Q such that:

(1) Q returns a subset of T.

(2) The WHERE clause of Q contains filters that correspond to the elements of the property value pairs of the nucleuses in $\boldsymbol{N}$.

(3) The WHERE clause of Q contains equijoin clauses that correspond to the edges in ST.

Section 4.2 illustrates the synthesis of SPARQL queries.

To conclude, we state a lemma that captures the correctness of the algorithm:

Lemma 2: Let T be an RDF dataset, S be the RDF schema of T and K be a keyword-based query. Let Q be the SPARQL query the translation algorithm outputs for K, T and S. Then, any result of Q is an answer for K over T with a single connected component.

Proof Sketch

Step 1 of translation algorithm computes all possible matches between keywords in K and the RDF dataset T. Step 2 constructs the nucleuses by combining the matches found in Step 1. Steps 3 and 4 create a set of nucleuses $\boldsymbol{N}$ such that $\boldsymbol{N}$ covers as many keywords as possible. Let $\boldsymbol{C_N}$ be the set of the classes of the nucleuses in $\boldsymbol{N}$. Note that $\boldsymbol{C_N}$ can be viewed as a set of nodes of the RDF schema diagram D_S. Step 5 connects, as much as possible, the nodes in $\boldsymbol{C_N}$ by paths in D_S, generating a Steiner tree *ST* of D_S that covers all nodes in $\boldsymbol{C_N}$. Finally, let Q be the CONSTRUCT query synthesized in Step 6 and A be a result of Q. Then, A is a subset of T and, by the construction of $\boldsymbol{N}$ and the filters in the WHERE clause of Q, A matches as many keywords in K as possible. Hence, A is answer for K over T. Furthermore, since *ST* is a Steiner tree of D_S that covers all nodes in $\boldsymbol{C_N}$, by the construction of the equijoin clauses in the WHERE clause of Q, the result A of Q will have a single connected component. □

4.2 An Example the Translation Process

This section illustrates how the algorithm synthesizes a SPARQL query for the following keyword-based query K:

```
Well Submarine Sergipe Vertical Sample
```

Step 1 searches the auxiliary tables ClassTable, PropertyTable and ValueTable to find matches with the keywords in K. For example, the following SQL query processes `Sergipe` against the ValueTable auxiliary table, whose columns are Property, Domain and Value ("fuzzy" is an Oracle function):

```
SELECT DISTINCT Property
FROM ValueTable
WHERE CONTAINS (Value, 'fuzzy({sergipe}, 70, 1)', 1) > 0
```

For the industrial dataset, Step 1 returns the following matches:

- A class metadata match
  M_1 = (`Sample`, (Sample, rdfs:label, "Sample"))
- A class metadata match
  M_2 = (`Well`, (DomesticWell, rdfs:label, "Domestic Well"))
- A property value match
  M_3 = (`Vertical`, (s, DomesticWell#Direction, v))
  since `Vertical` matches some value v of property DomesticWell#Direction (with domain DomesticWell).
- Two property value matches
  M_4 = (`Sergipe`, (s', DomesticWell#Location, v'))
  M_5 = (`Submarine`, (s'', DomesticWell#Location, v''))
  since `Submarine` and `Sergipe` match some values v' and v'' of property DomesticWell#Location (with domain DomesticWell).

Step 2 then generates two nucleuses:

- A first nucleus with just class Sample, using match M_1:
  N_1 = (({`Sample`}, Sample), ∅, ∅)
- A second nucleus with class DomesticWell, using match M_2, and a property value list using matches M_3, M_4 and M_5:
  N_2 = (({`Well`}, DomesticWell), ∅,
  {({`Vertical`}, DomesticWell#Direction),
  ({`Sergipe`, `Submarine`}, DomesticWell#Location)})

Step 3 computes the scores of nucleuses N_1 and N_2 as follows. The score of nucleus N_1 is simple the score of the match of the keyword `Sample` with the value of the class label, which is the string "Sample". The score of nucleus N_2 is given by:

$$score(N_2) = (\alpha s_C + \beta s_P + (1 - \alpha - \beta) s_V)$$

where

- s_C = *meta_sim*(({`Well`}, DomesticWell)), which is the score of the match of the keyword `Well` with the value of the class label, which is the string "Domestic Well"
- s_P = 0, since the property list of the nucleus is empty
- s_V = *value_sim*(({`Vertical`}, DomesticWell#Direction)) +
  value_sim(({`Submarine`, `Sergipe`},
  DomesticWell#Location))

For example, the value of

value_sim(({`Submarine`, `Sergipe`}, DomesticWell#Location))

is estimated by the following SQL query over the ValueTable auxiliary table, whose columns again are Property, Domain and Value (the prefix "ex:" is fictitious to preserve confidentiality of the data and "fuzzy" and "accum" are Oracle functions):

```
SELECT
   SCORE(1)/LENGTH(REGEXP_REPLACE(Value,'[^a-zA-Z0-9 -]',''))
   as score
FROM ValueTable
WHERE
    Domain = 'ex:DomesticWell' AND
    Property = ‘ex:DomesticWell#Location’ AND
    CONTAINS (Value,
         'fuzzy({submarine}, 70, 1) accum fuzzy({sergipe}, 70, 1)', 1) > 0
ORDER BY score DESC
OFFSET 0 ROWS FETCH NEXT 1 ROWS ONLY
```

Step 4 then selects the two nucleuses and Step 5 constructs a simple Steiner tree with just two nodes, corresponding to classes Sample and DomesticWell, connected by one edge, labelled with the object property Sample#DomesticWellCode.

Step 6 generates the SPARQL query Q below (which again uses the fictitious prefix "ex:"):

```
1.  SELECT  ?C0 ?C1 ?P0 ?P1
2.   (<http://xmlns.oracle.com/rdf/textScore>(1) AS ?score1)
3.   (<http://xmlns.oracle.com/rdf/textScore>(2) AS ?score2) .
4.  WHERE
5.   { ?I_C1 <ex:Sample#DomesticWellCode> ?I_C0 .
6.     ?I_C0 <ex:DomesticWell#Direction> ?P0 .
7.     ?I_C0 <ex:/DomesticWell#Location> ?P1
8.     FILTER (http://xmlns.oracle.com/rdf/textContains(?P0,
9.                                              "fuzzy({vertical}, 70, 1)", 1)
10.     || http://xmlns.oracle.com/rdf/textContains(?P1,
11.           "fuzzy({submarine}, 70, 1) accum fuzzy({sergipe}, 70, 1)", 2))
12.    ?I_C0 rdfs:label ?C0 .
13.    ?I_C1 rdfs:label ?C1
14.  }
15. ORDER BY DESC(?score1 + ?score2)
16. LIMIT   750
```

The TARGET clause in Line 1 returns a table with variable bindings (the SELECT form of the query results). Although we adopted the CONSTRUCT form of a query, which returns a set of triples, to explain the notion of a keyword-based query answer, users preferred to see the results as a table, as discussed in Section 4.3.

Step 6 constructs the WHERE clause of the SPARQL query as follows. The (only) edge of the Steiner tree, labelled with the object property Sample#DomesticWellCode, generates the triple pattern in Line 5. Note that, since the domain of Sample#DomesticWellCode is the class Sample and the range is the class DomesticWell, variables ?I_C1 and ?I_C0 will respectively bind to instances of these classes. Hence, it is not necessary to include triple patterns that force ?I_C1 to be of type Sample and ?I_C0 to be of type DomesticWell.

The property value list of nucleus N_2 generates the triple patterns in Lines 6 to 11. The triple pattern in Line 6 instantiates variable ?P0 with the value of property DomesticWell#Direction for instance ?I_C0. Likewise, the triple pattern in line 7 instantiates variable ?P1 with the value of property DomesticWell#Location for instance ?I_C0.

The FILTER declaration in lines 8 and 9 matches the keyword `Vertical` with the value in ?P0, using the Oracle fuzzy matching function with the appropriate parameters (70 and 1). The matching score is returned in the Oracle predefined variable ?score1 (which is indicated by the "1" that appears as the last parameter in line 9).

The FILTER declaration in Lines 10 and 11 matches one of the keywords `Submarine` or `Sergipe`, or both, with the value in ?P1, using the Oracle fuzzy matching function, with the appropriate parameters (70 and 1), and the accum parameter, to sum the matching scores, if indeed both keywords match the value in ?P1. The matching score is returned in the Oracle predefined variable ?score2 (which is indicated by the "2" that appears as the last parameter in line 11).

Lines 12 and 13 translate the URIs in ?l_C0 and ?l_C1 to labels, which are hopefully user-friendly, and bind them to ?C0 and ?C1.

Finally, lines 15 and 16 order the query results in descending order of the combined scores and limit the result to 750 lines.

4.3 User Interface

The user interface offers an auto-completion feature to help users formulate a keyword-based query, as in Figure 3a. The interface suggests new keywords based on the previous keywords, the RDF schema vocabulary, and the labels that are resource identifiers (such as the "Sergipe", the name of a state).

Since an answer *A* for a keyword-based query *K* over an RDF dataset *T* is formally a subset of *T*, it would be consistent to present *A* as a set of triples. However, this option proved to be inconvenient for the users, which are more familiar with tabular data, as in relational systems. We then implemented a user interface that presents the results of *K* by combining a table with the Steiner tree underlying the SPARQL query, as in Figure 3b. The user may also select additional properties to be included in the table, as in Fig. 3c.

Finally, the interface allows the user to specify a keyword-based query which includes *filters*, such as:

```
Sample with Top between 2000m and 3000m
```

A *simple filter* involves only comparison operators, expressed in symbolic form, such as "<", or using reserved words, such as "`between`", whereas a *complex filter* is a Boolean combination of simple filters, expressed using Boolean operators. A filter typically involves constants, perhaps with a unit of measure, such as "`2000m`"; the tool converts all constants to the unit of measure adopt for the property being filtered. The syntax of the filters is specified by a grammar defined in ANTLR4 (ANother Tool for Language Recognition) [16].

5. EXPERIMENTS

5.1 Experiment setup

All experiments were conducted using a RESTful Web application develop in Java. The app ran on a desktop machine with OS Windows 7 Ultimate, a quad-core processor Intel(R) Core(TM) i5-2450M CPU @ 2.50GHz, 4 GB of RAM. To store and manage the RDF data, we used the Oracle Spatial and Graph for Semantic Technologies of Oracle 12c [14], running on a quad-core machine with processor Intel(R) Core(TM) i5 CPU 660 @ 3.33GHz, 7GB of RAM, and 4096 KB of Cache size. The database was configured with a PGA size of 324 MB and an SGA size of 612 MB with 148 MB of cache size and 296 MB of buffer cache.

The label and description columns of the auxiliary tables (see Section 4.1) were indexed using the CREATE INDEX statement of Oracle Text [19] to facilitate full text search over the stored values.

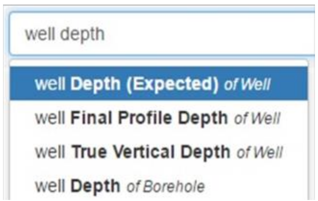

well depth
well Depth (Expected) of Well
well Final Profile Depth of Well
well True Vertical Depth of Well
well Depth of Borehole

Figure 3a. Example of auto-completion.

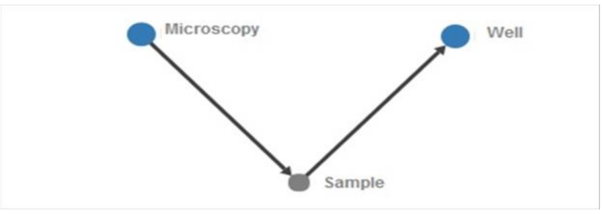


Figure 3b. Example of a query graph.

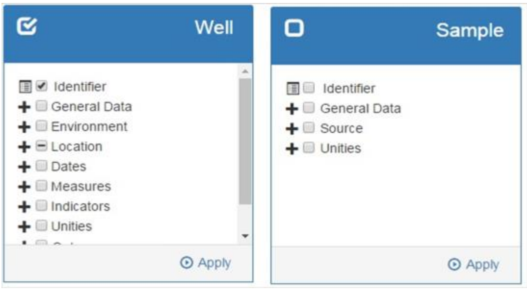


Figure 3c. Selection of additional properties.

In the case of RDF data, the Semantic Network feature of Oracle allows B-Tree indexing for RDF models and entailments [23].

5.2 Experiments with the Industrial Dataset

The data was originally stored in a conventional relational database, with well-documented tables and columns, which proved to be very helpful to identify metadata matches. The relational schema was normalized, as usual, which implies that a single table may represent several concepts and properties.

The triplification process used R2RML, the W3C standard RDB to RDF Mapping Language [6]. However, we soon realized that we had to capture additional metadata, such as which table columns were keys, which contained external names for the objects (such as state names and acronyms), etc. These additional metadata were important to guide keyword matches and to define how the object IDs were exposed to the users. Therefore, we proceeded as follows. First, on the relational side, we defined a set of views that denormalize the tables. Then, we created an XML document that defines all classes and properties of the RDF schema, as well as additional details, and that maps the RDF classes and properties one-to-one to the relational views. We developed a module that, using the XML document, generates the R2RML statements to map the relational data to triples and to load the auxiliary tables mentioned in Section 4.1.

Figure 4 shows a partial RDF schema diagram. The diagram depicts all classes (in rectangles), object properties (in single arrows, starting on the domain and ending on the range), with their names omitted to avoid cluttering the diagram, and subClassOf axioms (in dashed arrows, starting on the sub-class and ending on the super-class).

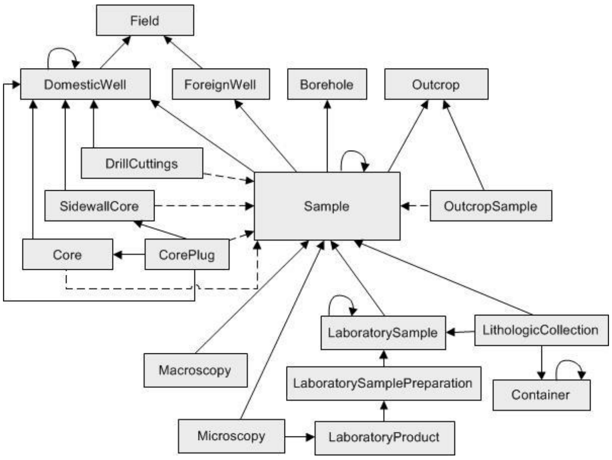


Figure 4. RDF schema of the industrial dataset.

The dataset is about hydrocarbon exploration (that is, oil and gas exploration). The instances of the central class, Sample, describe geological samples obtained during well drilling or directly from outcrops (rock formations visible on the surface). The instances of the classes DrillCuttings, SidewallCore, Core, CorePlug and OutcropSample correspond to sample sub-classes. The instances of the classes at the bottom of the diagram represent laboratory products, their macroscopic and microscopic analysis and where the products are stored.

The dataset has a large number of datatype properties (558). In particular, the values of the datatype properties of the instances of the classes Macroscopy and Microscopy are mostly literals, with a rich description of the laboratory products, which are highly amenable to keyword search. In fact, this motivated the project since users of the original relational database were mostly geologists, which were not happy with the relational database interface.

It took, on the average, 3 hours to triplify the relational database, generating an RDF dataset with about 130M triples (see Table 1), which implies that it is feasible to fully rematerialize the RDF dataset when needed, although we could have implemented an incremental rematerialization strategy.

We ran a suite of keyword-based queries to assess the performance of the tool, the correctness of the translation of the keyword-based queries and the adequacy of the result ranking. Table 2 (at the end of the paper) shows the runtime to process the keyword-based queries up until the first 75 answers were sent to the user, which corresponds to the first Web page (the time reported is the average of 10 executions for each sample query). The results show that all queries were successfully executed in less than 0.5 sec, which is quite reasonable, considering the size of the dataset.

Table 1. Statistics – Industrial dataset, IMDb and Mondial.

Triple Type	#Triples		
	Industrial	IMDb	Mondial
Class declarations	18	21	40
Object property declarations	26	24	62
Datatype property declarations	558	24	130
subClassOf axioms	5	-	-
Indexed properties	413	34	71
Distinct indexed prop instances	7.103.544	14.259.846	11.094
Class instances	8.981.679	72.973.275	43.869
Object property instances	11.072.953	184.818.637	63.652
Total triples	**130.058.210**	**395.394.424**	**235.387**

Finally, as a very preliminary user assessment, before early deployment, we asked 2 questions to 3 geologists to evaluate the same set of keyword-based queries. The results were very encouraging:

Question 1 (Correctness of the translation): “The results returned are a correct answer for the keyword-based query?”

Results: 8 x “Very Good”, 9 x “Good” and 1 x “Regular.

Question 2 (Adequacy of the ranking of the results): “The expected results appear in the first Web page returned?”

Results: 6 x “Very Good”, 11 x “Good” and 1 x “Regular.

Both “Regular” ratings were given by one of the users to the keyword-based query “`field exploration macroscopy microscopy lithologic collection`”, which is fairly generic and returns a large number of answers.

We also opened the tool to a small user community, all of whom were quite surprised with the ease of use of the tool and the quality of the answers, and manifested their interest in expanding the tool to other Petrobras databases, which attests the success of the project.

5.3 Experiments with Mondial and IMDb

We tested the tool against triplified versions of the Mondial dataset (https://www.dbis.informatik.uni-goettingen.de/Mondial/) and IMDb (https://sites.google.com/site/ontopiswc13/home/imdb-mo). Contrasting with the versions adopted in Coffman’s benchmark [4], these versions feature conceptual schemas with a complexity closer to the schema of the target industrial dataset (see Table 1). We used the same list of keyword queries as in Coffman’s benchmark, albeit they are much simpler than those expected from the typical users of the industrial dataset. We ran all queries against each of these datasets and compared the results returned with the expected results (the full results are available at www.inf.puc-rio.br/~casanova/ under the Recent Tools section).

A summary of the results for the Mondial RDF dataset follows:

Queries 1-5 – countries: All queries correctly answered.

Queries 6-10 – cities: Queries correctly answered, except Query 6, which returned 2 results, since there are 2 cities named “Alexandria”.

Query 11-15 – geographical: Queries correctly answered, except Query 12, which returned 2 results, since “Niger” is both a country and a river.

Queries 16-20 – organization: Some queries were not correctly answered since the expected values were not listed in class Organization (in the version of Mondial used).

Queries 21-25 – border between countries: Keywords match the labels of two instances of class Country; but the keywords are not sufficient to infer that the question is about the borders between countries and, thus, were not correctly answered.

Queries 26-35 – geopolitical or demographic information: Queries correctly answered, except Query 32.

Queries 36-45 – member organizations two countries belong to: The expected answer is the list of organizations that the countries belong to; however, the translation algorithm did not identify the IS_MEMBER class when generating the nucleuses.

Queries 46-50 – Miscellaneous: Some queries were successfully answered, while others were not, since the keywords do not always reflect the intended question.

A total of 32 queries, 64% of the 50 queries in Coffman's benchmark for Mondial, were correctly answered. As pointed out above, an analysis of the failed queries (see Table 3 for examples) reveals that: some may not be classified as failures (failed queries in the first 20 queries); some can be blamed to the lack of keyword semantics (failed queries in groups *21-25* and *46-50*); and some to the lack of accuracy of the keywords (as Query 50 in Table 3). These results actually indicate that the list of queries and query results in Coffman's benchmark should be reassessed.

Table 4 (also at the end of the paper) reports the results for the IMDb dataset. A total of 36 queries, 72% of the 50 queries in Coffman benchmark for IMDb, were correctly answered. Again, an analysis of the failed queries is instructive. For example, when running Query 41, we found a 1951 film with "Audrey Hepburn" in the title, rather than all 1951 films that the actress Audrey Hepburn starred. However, we would rather classify this result as a serendipitous discovery, rather than a failure.

6. CONCLUSIONS

We presented the results of an industrial project to facilitate access to a large relational database by combining RDF technology with keyword search. The algorithm to translate keyword-based queries to SPARQL queries takes advantage of the schema of the RDF dataset to avoid user intervention and achieve good performance, even for large RDF datasets. The user interface allows the user to specify keywords, as well as filters and unit measures. The interface presents the result of a keyword query with the help of tables, which is a familiar form of expressing query results, rather than as an RDF graph. Finally, the experiments covered both a real-world industrial dataset, as well as two familiar benchmarks. The tool proved to be quite robust to keyword-based queries over datasets with complex conceptual schemas, and not just toy schemas, which encourages its wider adoption at Petrobras, the industrial partner that supported the project reported in this paper.

As for future work, we plan to incorporate a domain ontology, being developed as a separated project, to expand keywords and therefore improve the usefulness of the tool. We also plan to allow filters with spatial operators. Lastly, we are working on a version of the application for a dataset federation.

7. ACKNOWLEDGMENTS

This work was partly funded by CNPq under grant 444976/2014-0, 303332/2013-1, 442338/2014-7 and 248743/2013-9 and by FAPERJ under grant E-26/201.337/2014. We would also like to thank the Petrobras team who help the authors design the tool.

8. REFERENCES

[1] Aditya, B., Bhalotia, G., Chakrabarti, S., Hulgeri, A., Nakhe, C., Parag, P., and Sudarshan, S. 2002. BANKS: Browsing and keyword searching in relational databases. VLDB 2002, 1083-1086.

[2] Agrawal, S., Chaudhuri, S. and Das, G. 2002. DBXplorer: A system for keyword-based search over relational databases. ICDE 2002, 5-16.

[3] Brickley, D. and Guha, R.V. (eds). 2014. RDF Schema 1.1. W3C Recommendation 25 February 2014.

[4] Coffman, J. and Weaver, A. 1999. An empirical performance evaluation of relational keyword search techniques. TKDE 1999.

[5] Cyganiak, R., David Wood, D. and Lanthaler, M. (eds.). 2014. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation (25 February 2014).

[6] Das, S., Sundara, S. and Cyganiak, R. (eds). 2012. R2RML: RDB to RDF Mapping Language. W3C Recommendation 27 September 2012.

[7] De Virgilio, R. 2012. RDF Keyword Search Query Processing via Tensor calculus. WWW 2012.

[8] De Virgilio, R. and Maccioni. A. 2014. Distributed Keyword Search over RDF via MapReduce. ESWC 2014, 208-223.

[9] Guo, Y., Pan, Z. and Heflin, J. 2004. LUBM: A Benchmark for OWL Knowledge Base Systems. J. Web Semantics 3(2-3), 158-182.

[10] Harris, S. and Seaborne, A. 2013. SPARQL 1.1 Query Language. W3C Recommendation 21 March 2013.

[11] He, H., Wang, H., Yang, J. and Yu, P. 2007. Blinks: Ranked keyword searches on graphs. ACM SIGMOD 2007, 305-316.

[12] Hristidis, V. and Papakonstantinou, Y. 2002. DISCOVER: keyword search in relational databases. VLDB 2002, 670-681.

[13] Lanti, D., Rezk, M., Xiao, G. and Calvanese, D. 2015.The NPD Benchmark: Reality Check for OBDA Systems. EDBT 2015 – Industry and Applications.

[14] Murray, C. 2014. *Spatial and Graph RDF Semantic Graph Developer's Guide 12c*. Oracle. p. 636. E51611-06.

[15] Oliveira, P., Silva, A. and Moura, E. 2015. Ranking Candidate Networks of relations to improve keyword search over relational databases. ICDE 2015, 399-410

[16] Parr, T. 2013. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

[17] Schmachtenberg, M., Bizer, C. and Paulheim, H. 2014. State of the LOD Cloud 2014 (Version 0.4, 08/30/2014) (available at: http://lod-cloud.net).

[18] Qiao, S. and Özsoyoğlu, Z. 2015. RBench: Application-Specific RDF Benchmarking. SIGMOD 2015, 1825-1838.

[19] Shea, C. 2014. *Oracle Text Reference, 12c Release 1 (12.1)*. Oracle. p. 718. E41399-05.

[20] Skjæveland, M., Giese, M., Hovland, D., Lian, E. and Waaler, A. 2015. Engineering ontology-based access to real-world data sources. J. Web Semantics: 33, 112-140.

[21] Tran, T., Wang, H., Rudolph, S. and Cimiano, P. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. ICDE 2009, 405-416.

[22] Vidal, V., Casanova, M., Neto, L.E. and Monteiro, J.M. 2014. A Semi-Automatic Approach for Generating Customized R2RML Mappings. ACM SAC 2014, 316-322.

[23] Wu, S., Perry, M. and Kolovski, V. 2010. *Oracle Database Semantic Technologies: Understanding How to Install, Load, Query and Inference*. Oracle.

[24] Yang, M., Ding, B., Chaudhuri, S. and Chakrabarti, K. 2014. Finding patterns in a knowledge base using keywords to compose table answers. VLDB Endow. 7(14), 1809-1820.

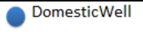
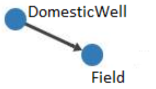
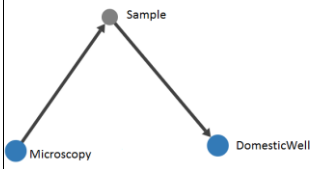
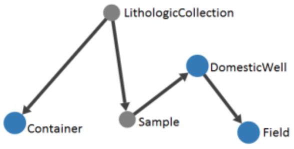
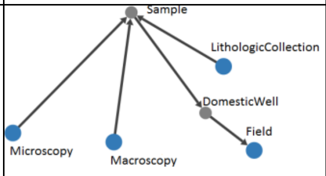
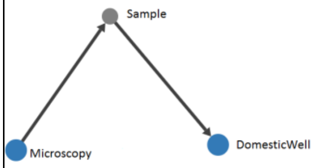
[25] Zenz, G., Zhou, X., Minack, E., Siberski, W. and Nejdl, W. 2009. From keywords to semantic queries - incremental query construction on the semantic web. J. Web Semantics 7(3), 166-176.

[26] Zhang, L., Tran, T. and Rettinger, A. 2013. Probabilistic Query Rewriting for Efficient and Effective Keyword Search on Graph Data. VLDB 2013, 1642-1653.

[27] Zheng, W., Zou, L., Peng, W., Yan, X., Song, S. and Zhao, D. 2016. Semantic SPARQL similarity search over RDF knowledge graphs. VLDB Endow. 9(11), 840-851.

[28] Zhou, Q., Wang, C., Xiong, M., Wang, H. and Yu, Y. 2007. SPARK: Adapting keyword query to semantic search. ISWC, 2007.

Table 2. Runtime to process sample keyword-based queries.

Keywords	Structure of the SPARQL query	Description of the nucleuses	Elapsed time (in milliseconds)		
			Query Synthesis	Query Execution	Total*
`well sergipe`		• A single nucleus with class DomesticWell • `sergipe` matches values of properties Basin, Localization, Federation, among others, of DomesticWell	15,4	446,3	**462,0**
`well salema`		• Two nucleuses with classes DomesticWell and Field, where the first one matches `well` • `salema` matches values of property Name of Field	25,0	246,4	**271,6**
`microscopy well sergipe`		• Two nucleuses with classes DomesticWell and Microscopy, which match the first two keywords • `sergipe` matches values of properties Localization, Basin, Federation, among others, of DomesticWell • The path from Microscopy to DomesticWell goes through the class Sample	23,2	327,3	**350,8**
`container well field salema`	 LithologicCollection DomesticWell Container Sample Field	• The first three keywords respectively match classes Container, DomesticWell and Field • `salema` matches values of property Name of Field • The non-directed path to join Container with DomesticWell and Field goes through Sample and LithologicCollection	24,3	315	**339,5**
`field exploration macroscopy microscopy lithologic collection`	 Sample LithologicCollection DomesticWell Field Microscopy Macroscopy	• `exploration` matches values of properties OperativeUnit and AdministrativeUnit of class Field • According to the order they appear, the other keywords match classes Field, Macroscopy, Microscopy, and LithologicColletion • The paths leaving from Macroscopy, Microscopy, and LithologicColletion to Field go through the classes Sample and DomesticWell	43,8	180,1	**224,1**
`well coast distance < 1km microscopy bio-accumulated cadastral date between October 16, 2013 and October 18, 2013`		• Two nucleuses with classes DomesticWell and Microscopy • `coast distance` is a property of class DomesticWell filtered by the condition "`< 1km`" • `bio-accumulated` matches property Name of Microscopy • `cadastral date` is a property of class Microscopy, whose data type is date and is subjected to a filter • The path from Microscopy to DomesticWell goes through the class Sample	95,4	108,4	**204,1**

(*) Up to sending the first 75 answers.

Table 3. Selected queries from the Mondial Benchmark.

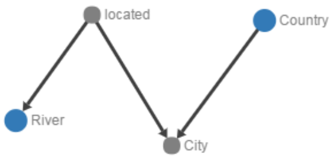
#Query	Keywords	Expected Answer	Application Answer	Observation
Query 16	`arab cooperation council`	Arab Cooperation Council	75 instances of class Organization	"Arab Cooperation Council" is not listed in class Organization (in the version of Mondial used)
Query 32	`Uzbekistan eastern orthodox`	Uzbekistan	-	"eastern orthodox" does not exist for property Name of class Religion (in the version of Mondial used)
Query 50: *Which Egyptian provinces does the Nile River flow through?*	`egypt nile`	Asyut Beni Suef El Giza El Minya El Qahira (munic.)	Egypt | Nile 	If the keyword `city` were added, we would correctly obtain: Egypt | Al Minya | Nile Egypt | Al Qahirah | Nile Egypt | Al Jizah | Nile Egypt | Bani Suwayf | Nile Egypt | Asyut | Nile

Table 4. Analysis of the IMDb Benchmark.

Queries 1-10: consist of the name of movie stars, such as “denzel washington”. Relevant results contain a single tuple from the person relation that is the tuple of the specified individual.
Accuracy: 10 of 10. The result contained more than one tuple, if the movie star’s name matched one of the keywords, but the top result was the expected actor.

Queries 11-20: consist of the name of movies, such as “gone with the wind”. Relevant results contain a single tuple from the title relation that is the tuple of the specified film.
Accuracy: **9 of 10**. Again, the result contained more than one tuple, if the movie name matched one of the keywords, but the top result was the expected movie.
Error in Query 13 – “casablanca”. “casablanca” is the name of a movie and of an actor; the score for both values was the same, but the algorithm returned the name of the actor, since the Actor class had a higher score than the Movies class. The movie name was the second generated query.

Queries 21-30: consist of the keyword “'title'” plus the name of film characters, such as “title atticus finch”. Relevant results contain 3 tuples (1 from the char_name relation, 1 from the cast_info relation, and 1 from the title relation) that link the character to the film(s) in which s/he appears. (The keyword “title” is intentionally added to differentiate this group of topics from topics 1-20)
Accuracy: **7 of 10.** Again, the result contained more than one tuple.
Error in Queries 22, 23. The name of the character is part of the name of some title. The nucleus with class Title contained all keywords and had the best score. The answers of the algorithm were the titles with the character names.
Error in Query 28. In this case, the class AKA_TITLE has “darth vader” in one of its values. This nucleus was the best scored because the label of the class had the keyword “title” and “darth vader” as a value. Class Title only matched the keyword “title” and class char_name only matched “darth vader”.

Queries 31-35: consist of the keyword “'title'” plus a film quote, such as “title frankly my dear i don't give a damn”. Relevant results contain 2 tuples (1 from the movie_info relation and 1 from the title relation) that link the movie quote to the film in which it appears. (The keyword “title” is intentionally added so that relevant results answer the question "In which film does this quote appear?".) Note that a quote may appear in multiple films.
Accuracy: 4 of 5. The result was not a single tuple, as in previous blocks.
Error in Query 32: The quotes were not in the dataset used for the tests.

Query 36 “mark hamill luke skywalker”. Relevant results must denote the films in which the actor Mark Hamill plays the character Luke Skywalker.
Accuracy: 1 of 1

Query 37 “tom hanks 2004”: Relevant results contain 3 tuples (name <- cast_info -> title) that must denote all films in which the actor Tom Hanks appeared in the year 2004.
Accuracy: 1 of 1

Queries 38-40: Relevant results must denote the character that an actor plays in a film, such as “henry fonda yours mine ours char_name”
Accuracy: 1 of 3
Error in Queries 38 and 39: There are values in char_name that match “Henry Fonda” and “Russell Crowe”. The algorithm assumed that the query was about these character names and tested with the movie name.

Query 41 “audrey hepburn 1951”: Relevant results contain 3 tuples (name <- cast_info -> title) that must denote all films in which the actor Audry Hepburn appeared in the year 1951.
Accuracy: 0 of 1
Error: The nucleus with Title covered all three keywords since there is a film whose name matches “Audrey Hepburn” and whose production year matches 1951.

Query 42 “name jacques clouseau”: A relevant result must identify an actor who plays Jacques Clouseau in a movie.
Accuracy: 0 of 2
Error: The algorithm found only the nucleus with class char_name, the character name matched with property name, and the keyword “name” matched with the label of the nucleus.

Query 44 “rocky stallone”: Relevant results must denote a film in which Sylvester Stallone plays the character Rocky. Note that because of limitations of existing systems, relevant results are *not* required to include the appropriate tuple from the title relation (which would prevent any system from identifying a single relevant result).
Accuracy: 0 of 1
Error: the keywords are very ambiguous. The algorithm found both keywords in a PERSON_INFO#INFO value.

Query 45 “name terminator”: A relevant result must identify an actor who plays "The Terminator"
Accuracy: 0 of 1
Error: same as for Queries 42-43.

Queries 46-49: Relevant results identify relationships (through the title relation) between an actor and another class, such as “harrison ford george lucas”.
Accuracy: 3 of 4
Error in Query 48: “wachowski” only had matches in the AKA_NAME class.

Query 50 “indiana jones last crusade lost ark”: Relevant results identify cast members in common between the films "Raiders of the Lost Ark" and "Indiana Jones and the Last Crusade."
Accuracy: 0 of 1
Error: The algorithm did not return the actors that both movies had in common, but returned the movies themselves.

Hi-WAY:
Execution of Scientific Workflows on Hadoop YARN

Marc Bux
Humboldt-Universität zu Berlin
Berlin, Germany
bux@informatik.hu-berlin.de

Jörgen Brandt
Humboldt-Universität zu Berlin
Berlin, Germany
joergen.brandt@hu-berlin.de

Carl Witt
Humboldt-Universität zu Berlin
Berlin, Germany
wittcarx@informatik.hu-berlin.de

Jim Dowling
Swedish Institute of Computer Science (SICS)
Stockholm, Sweden
jdowling@sics.se

Ulf Leser
Humboldt-Universität zu Berlin
Berlin, Germany
leser@informatik.hu-berlin.de

ABSTRACT

Scientific workflows provide a means to model, execute, and exchange the increasingly complex analysis pipelines necessary for today's data-driven science. However, existing scientific workflow management systems (SWfMSs) are often limited to a single workflow language and lack adequate support for large-scale data analysis. On the other hand, current distributed dataflow systems are based on a semi-structured data model, which makes integration of arbitrary tools cumbersome or forces re-implementation. We present the scientific workflow execution engine Hi-WAY, which implements a strict black-box view on tools to be integrated and data to be processed. Its generic yet powerful execution model allows Hi-WAY to execute workflows specified in a multitude of different languages. Hi-WAY compiles workflows into schedules for Hadoop YARN, harnessing its proven scalability. It allows for iterative and recursive workflow structures and optimizes performance through adaptive and data-aware scheduling. Reproducibility of workflow executions is achieved through automated setup of infrastructures and re-executable provenance traces. In this application paper we discuss limitations of current SWfMSs regarding scalable data analysis, describe the architecture of Hi-WAY, highlight its most important features, and report on several large-scale experiments from different scientific domains.

1. INTRODUCTION

Recent years have brought an unprecedented influx of data across many fields of science. In genomics, for instance, the latest generation of genomic sequencing machines can handle up to 18,000 human genomes per year [41], generating about 50 terabytes of sequence data per week. Similarly, astronomical research facilities and social networks are also generating terabytes of data per week [34]. To synthesize succinct results from these readouts, scientists assemble complex graph-structured analysis pipelines, which chain a multitude of different tools for transforming, filtering, and aggregating the data [18]. The tools used within these pipelines are implemented by thousands of researchers around the world, rely on domain-specific data exchange formats, and are updated frequently (e.g., [27, 31]). Consequently, easy ways of assembling and altering analysis pipelines are of utmost importance [11]. Moreover, to ensure reproducibility of scientific experiments, analysis pipelines should be easily sharable and execution traces must be accessible [12].

Systems fulfilling these requirements are generally called scientific workflow management systems (SWfMSs). From an abstract perspective, scientific workflows are compositions of sequential and concurrent data processing tasks, whose order is determined by data interdependencies [36]. Tasks are treated as black boxes and can therefore range from a simple shell script over a local command-line tool to an external service call. Also, the data exchanged by tasks is typically not parsed by the SWfMS but only forwarded according to the workflow structure. While these black-box data and operator models prohibit the automated detection of potentials for data-parallel execution, their strengths lie in their flexibility and the simplicity of integrating external tools.

To deal with the ever-increasing amounts of data prevalent in today's science, SWfMSs have to provide support for parallel and distributed storage and computation [26]. However, while extensible distributed computing frameworks like Hadoop YARN [42] or MESOS [19] keep developing rapidly, established SWfMSs, such as Taverna [47] or Pegasus [13] are not able to keep pace. A particular problem is that most SWfMSs tightly couple their own custom workflow language to a specific execution engine, which can be difficult to configure and maintain alongside other execution engines that are already present on the cluster. In addition, many of these execution engines fail to keep up with the latest developments in distributed computing, e.g., by storing data in a central location, or by neglecting data locality and heterogeneity of distributed resources during workflow

Series ISSN: 2367-2005

10.5441/002/edbt.2017.87

scheduling [10]. Furthermore, despite reproducibility being advocated as a major strength of scientific workflows, most systems focus only on sharing workflows, disregarding the provisioning of input data and setup of the execution environment [15, 33]. Finally, many systems severely limit the expressiveness of their workflow language, e.g., by disallowing conditional or recursive structures. While the scientific workflow community is becoming increasingly aware of these issues (e.g., [8, 33, 50]), to date only isolated, often domain-specific solutions addressing only subsets of these problems have been proposed (e.g., [6, 14, 38]).

At the same time, support for many of these features has been implemented in several recently developed distributed dataflow systems, such as Spark [49] or Flink [5]. However, such systems employ a semi-structured white-box (e.g., key-value-based) data model to automatically partition and parallelize dataflows. Unfortunately, a structured data model impedes the flexibility in workflow design when integrating external tools that read and write file-based data. To circumvent this problem, additional glue code for transforming to and from the structured data model has to be provided. This introduces unnecessary overhead in terms of time required for implementing the glue code as well as for the necessary data transformations at runtime [48].

In this application paper, we present the **Hi**-WAY **W**orkflow **A**pplication master for **Y**ARN. Technically, Hi-WAY is yet another application master for YARN. Conceptually, it is a (surprisingly thin) layer between scientific workflow specifications expressed in different languages and Hadoop YARN. It emphasizes data center compatibility by being able to run on YARN installations of any size and type of underlying infrastructure. Compared to other SWfMSs, Hi-WAY brings the following specific features.

1. *Multi-language* support. Hi-WAY employs a generic yet powerful execution model. It has no own specification language, but instead comes with an extensible language interface and built-in support for multiple workflow languages, such as Cuneiform [8], Pegasus DAX [13], and Galaxy [17] (see Section 3.2).

2. *Iterative* workflows. Hi-WAY's execution model is expressive enough to support data-dependent control-flow decisions. This allows for the design of conditional, iterative, and recursive structures, which are increasingly common in distributed dataflows (e.g., [28]), yet are just beginning to emerge in scientific workflows (see Section 3.3).

3. *Performance* gains through *adaptive* scheduling. Hi-WAY supports various workflow scheduling algorithms. It utilizes statistics of earlier workflow executions to estimate the resource requirements of tasks awaiting execution and exploit heterogeneity in the computational infrastructures during scheduling. Also, Hi-WAY supports adaption of schedules to both data locality and resource availability (see Section 3.4).

4. *Reproducible* experiments. Hi-WAY generates comprehensive provenance traces, which can be directly re-executed as workflows (see Section 3.5). Also, Hi-WAY uses Chef [2] and Karamel [1] for specifying automated setups of a workflow's software requirements and input data, including (if necessary) the installation of Hi-WAY and Hadoop (see Section 3.6).

5. *Scalable* execution. By employing Hadoop YARN as its underlying execution engine, Hi-WAY harnesses its scalable resource management, fault tolerance, and distributed file management (see Section 3.1).

While some of these features have been briefly outlined in the context of a demonstration paper [9], this is the first comprehensive description of Hi-WAY.

The remainder of this paper is structured as follows: Section 2 gives an overview of related work. Section 3 presents the architecture of Hi-WAY and gives detailed descriptions of the aforementioned core features, which are highlighted in italic font throughout the rest of the document. Section 4 describes several experiments showcasing these feature in real-life workflows on both local clusters and cloud computing infrastructure. Section 5 concludes the paper.

2. RELATED WORK

Projects with goals similar to Hi-WAY can be separated into two groups. The first group of systems comprises traditional SWfMSs, which, like Hi-WAY, employ black-box data and operator models. The second group encompasses distributed dataflow systems developed to process mostly structured or semi-structured (white-box) data. For a comprehensive overview of data-intensive scientific workflow management, readers are referred to [10] and [26].

2.1 Scientific Workflow Management

The SWfMS Pegasus [13] emphasizes *scalability*, utilizing HTCondor as its underlying execution engine. It enforces the usage of its own XML-based workflow language called DAX. Pegasus supports a number of scheduling policies, all of which are static, yet some of which can be considered *adaptive* (such as HEFT [39]). Finally, Pegasus does not allow for *iterative* workflow structures, since every task invocation has to be explicitly described in the DAX file. In contrast to Hi-WAY, Pegasus does not provide any means of reproducing scientific experiments across datacenters. Hi-WAY complements Pegasus by enabling Pegasus workflows to be run on top of Hadoop YARN, as outlined in Section 3.2.

Taverna [47] is an established SWfMS that focuses on usability, providing a graphical user interface for workflow design and monitoring as well as a comprehensive collection of pre-defined tools and remote services. Taverna emphasizes *reproducibility* of experiments and workflow sharing by integrating the public myExperiment workflow repository [16], in which over a thousand Taverna workflows have been made available. However, Taverna is mostly used to integrate web services and short-running tasks and thus does not support *scalable* distribution of workload across several worker nodes or any *adaptive* scheduling policies.

Galaxy [17] is a SWfMS that provides a web-based graphical user interface, an array of built-in libraries with a focus on computational biology, and a repository for sharing workflows and data. CloudMan [3] extends Galaxy with limited *scalability* by enabling Galaxy clusters of up to 20 nodes to be set up on Amazon's EC2 through an easy-to-use web interface. Unfortunately, Galaxy neither supports *adaptive* scheduling nor *iterative* workflow structures. Similar to Pegasus and as described in Section 3.2, Hi-WAY complements Galaxy by allowing exported Galaxy workflows to be run on Hadoop YARN. For a comparative evaluation of Hi-WAY and Galaxy CloudMan, refer to Section 4.2.

Text-based parallel scripting languages like Makeflow [4], Snakemake [23], or Swift [45] are more light-weight alternatives to full-fledged SWfMSs. Swift [45] provides a functional scripting language that facilitates the design of inherently data-parallel workflows. Conversely, Snakemake [23] and Makeflow [4] are inspired by the build automation tool GNU make, enabling a goal-driven assembly of workflow scripts. All of these system have in common that they support the *scalable* execution of implemented workflows on distributed infrastructures, yet disregard other features typically present in SWfMSs, such as *adaptive* scheduling mechanisms or support for *reproducibility.*

Nextflow [38] is a recently proposed SWfMS [14], which brings its own domain-specific language. In Nextflow, software dependencies can be provided in the form of Docker or Shifter containers, which facilitates the design of *reproducible* workflows. Nextflow enables *scalable* execution by supporting several general-purpose batch schedulers. Compared to Hi-WAY, execution traces are less detailed and not re-executable. Furthermore, Nextflow does not exploit data-aware and *adaptive* scheduling potentials.

Toil [43] is a *multi-language* SWfMS that supports *scalable* workflow execution by interfacing with several distributed resource management systems. Its supported languages include the Common Workflow Language (CWL) [6], a YAML-based workflow language that unifies concepts of various other languages, and a custom Python-based DSL that supports the design of *iterative* workflows. Similar to Nextflow, Toil enables sharable and *reproducible* workflow runs by allowing tasks to be wrapped in re-usable Docker containers. In contrast to Hi-WAY, Toil does not gather comprehensive provenance and statistics data and, consequently, does not support any means of *adaptive* workflow scheduling.

2.2 Distributed Dataflows Systems

Distributed dataflow systems like Spark [49] or Flink [5] have recently achieved strong momentum both in academia and in industry. These systems operate on semi-structured data and support different programming models, such as SQL-like expression languages or real-time stream processing. Departing from the black-box data model along with natively supporting concepts like data streaming and in-memory computing allows these systems to in many cases execute even sequential processing steps in parallel and circumvent the materialization of intermediate data on the hard disk. It also enables the automatic detection and exploitation of potentials for data parallelism. However, the resulting gains in *performance* come at the cost of reduced flexibility for workflow designers. This is especially problematic for scientists from domains other than the computational sciences. Since integrating external tools processing unstructured, file-based data is often tedious and undermines the benefits provided by dataflow systems, a substantial amount of researchers continue to rely on traditional scripting and programming languages to tackle their data-intensive analysis tasks (e.g., [27, 31]).

Tez [32] is an application master for YARN that enables the execution of DAGs comprising map, reduce, and custom tasks. Being a low-level library intended to be interfaced by higher-level applications, external tools consuming and producing file-based data need to be wrapped in order to be used in Tez. For a comparative evaluation between Hi-WAY and Tez, see Section 4.1.

While Tez runs DAGs comprising mostly map and reduce tasks, Hadoop workflow schedulers like Oozie [20] or Azkaban [35] have been developed to schedule DAGs consisting mostly of Hadoop jobs (e.g., MapReduce, Pig, Hive) on a Hadoop installation. In Oozie, tasks composing a workflow are transformed into a number of MapReduce jobs at runtime. When used to run arbitrary scientific workflows, systems like Oozie or Azkaban either introduce unnecessary overhead by wrapping the command-line tasks into degenerate MapReduce jobs or do not dispatch such tasks to Hadoop, but run them locally instead.

Chiron [30] is a *scalable* workflow management system in which data is represented as relations and workflow tasks implement one out of six higher-order functions (e.g., map, reduce, and filter). This departure from the black-box view on data inherent to most SWfMSs enables Chiron to apply concepts of database query optimization to optimize *performance* through structural workflow reordering [29]. In contrast to Hi-WAY, Chiron is limited to a single, custom, XML-based workflow language, which does not support *iterative* workflow structures. Furthermore, while Chiron, like Hi-WAY, is one of few systems in which a workflow's (incomplete) provenance data can be queried during execution of that same workflow, Chiron does not employ this data to perform any *adaptive* scheduling.

3. ARCHITECTURE

Hi-WAY utilizes Hadoop as its underlying system for the management of both distributed computational resources and storage (see Section 3.1). It comprises three main components, as shown in Figure 1. First, the Workflow Driver parses a scientific workflow specified in any of the supported workflow languages and reports any discovered tasks to the Workflow Scheduler (see Sections 3.2 and 3.3). Secondly, the Workflow Scheduler assigns tasks to compute resources provided by Hadoop YARN according to a selected scheduling policy (see Section 3.4). Finally, the Provenance Manager gathers comprehensive provenance and statistics information obtained during task and workflow execution, handling their long-term storage and providing the Workflow Scheduler with up-to-date statistics on previous task executions (see Section 3.5). Automated installation routines for the setup of Hadoop, Hi-WAY, and selected workflows are described in Section 3.6.

3.1 Interface with Hadoop YARN

Hadoop version 2.0 introduced the resource management component YARN along with the concept of job-specific application masters (AMs), increasing scalability beyond 4,000 computational nodes and enabling native support for non-MapReduce AMs. Hi-WAY seizes this concept by providing its own AM that interfaces with YARN.

To submit workflows for execution, Hi-WAY provides a light-weight client program. Each workflow that is launched from a client results in a separate instance of a Hi-WAY AM being spawned in its own container. Containers are YARN's basic unit of computation, encapsulating a fixed amount of virtual processor cores and memory which can be specified in Hi-WAY's configuration. Having one dedicated AM per workflow results in a distribution of the workload associated with workflow execution management and is therefore required to fully unlock the *scalability* potential provided by Hadoop.

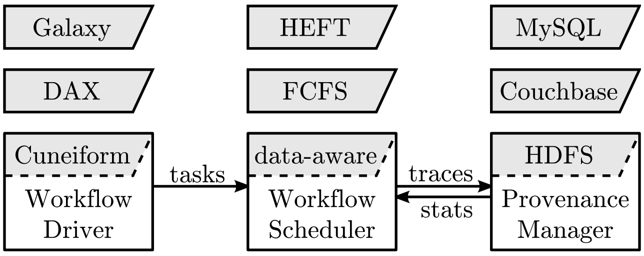


Figure 1: The architecture of the Hi-WAY application master: The Workflow Driver, described in Sections 3.2 and 3.3, parses a textual workflow file, monitors workflow execution, and notifies the Workflow Scheduler whenever it discovers new tasks. For tasks that are ready to be executed, the Workflow Scheduler, presented in Section 3.4, assembles a schedule. Provenance and statistics data obtained during workflow execution are handled by the Provenance Manager (see Section 3.5) and can be stored in a local file as well as a MySQL or Couchbase database.

For any of a workflow's tasks that await execution, the Hi-WAY AM responsible for running this particular workflow then requests an additional worker container from YARN. Once allocated, the lifecycle of these worker containers involves (i) obtaining the task's input data from HDFS, (ii) invoking the commands associated with the task, and (iii) storing any generated output data in HDFS for consumption by other containers executing tasks in the future and possibly running on other compute nodes. Figure 2 illustrates this interaction between Hi-WAY's client application, AM and worker containers, as well as Hadoop's HDFS and YARN components.

Besides having dedicated AM instances per workflow, another prerequisite for *scalable* workflow execution is the ability to recover from failures. To this end, Hi-WAY is able to re-try failed tasks, requesting YARN to allocate the additional containers on different compute nodes. Also, data processed and produced by Hi-WAY persists through the crash of a storage node, since Hi-WAY exploits the redundant file storage of HDFS for any input, output, and intermediate data associated with a workflow.

3.2 Workflow Language Interface

Hi-WAY sunders the tight coupling of scientific workflow languages and execution engines prevalent in established SWfMSs. For this purpose, its Workflow Driver (see Section 3.3) provides an extensible, *multilingual* language interface, which is able to interpret scientific workflows written in a number of established workflow languages. Currently, four scientific workflow languages are supported: (i) the textual workflow language Cuneiform [8], (ii) DAX, which is the XML-based workflow language of the SWfMS Pegasus [13], (iii) workflows exported from the SWfMS Galaxy [17], and (iv) Hi-WAY provenance traces, which can also be interpreted as scientific workflows (see Section 3.5).

Cuneiform [8] is a minimal workflow language that supports direct integration of code written in a large range of external programming languages (e.g., Bash, Python, R, Perl, Java). It supports *iterative* workflows and treats tasks as black boxes, allowing the integration of various tools and libraries independent of their programming API. Cuneiform facilitates the assembly of highly parallel data processing pipelines by providing a range of second-order functions extending beyond map and reduce operations.

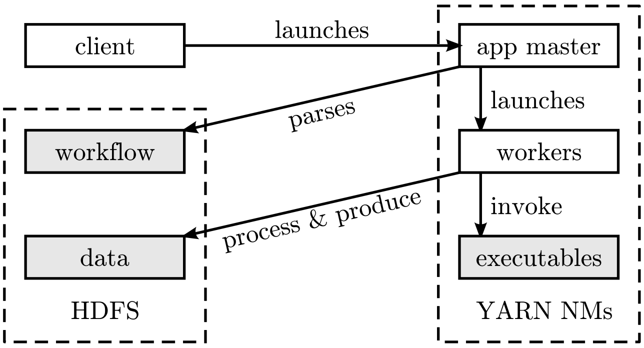


Figure 2: Functional interaction between the components of Hi-WAY and Hadoop (white boxes; see Section 3.1) as well as further requirements for running workflows (gray boxes; see Section 3.6). A workflow is launched from a client application, resulting in a new instance of a Hi-WAY AM within a container provided by one of YARN's NodeManagers (NMs). This AM parses the workflow file residing in HDFS and prompts YARN to spawn additional worker containers for tasks that are ready to run. During task execution, these worker containers obtain input data from HDFS, invoke locally available executables, and generate output data, which is placed in HDFS for use by other worker containers.

DAX [13] is Pegasus' built-in workflow description language, in which workflows are specified in an XML file. Contrary to Cuneiform, DAX workflows are static, explicitly specifying every task to be invoked and every file to be processed or produced by these tasks during workflow execution. Consequently, DAX workflows can become quite large and are not intended to be read or written by workflow developers directly. Instead, APIs enabling the generation of DAX workflows are provided for Java, Python, and Perl.

Workflows in the web-based SWfMS Galaxy [17] can be created using a graphical user interface, in which the tasks comprising the workflow can be selected from a large range of software libraries that are part of any Galaxy installation. This process of workflow assembly results in a static workflow graph that can be exported to a JSON file, which can then be interpreted by Hi-WAY. In workflows exported from Galaxy, the workflow's input files are not explicitly designated. Instead, input ports serve as placeholders for the input files, which are resolved interactively when the workflow is committed to Hi-WAY for execution.

In addition to these workflow languages, Hi-WAY can easily be extended to parse and execute other non-interactive workflow languages. For non-*iterative* languages, one only needs to extend the Workflow Driver class and implement the method that parses a textual workflow file to determine the tasks and data dependencies composing the workflow.

3.3 Iterative Workflow Driver

On execution onset, the Workflow Driver parses the workflow file to determine inferable tasks along with the files they process and produce. Any discovered tasks are passed to the Workflow Scheduler, which then assembles a schedule and creates container requests whenever a task's data dependencies are met. Subsequently, the Workflow Driver supervises workflow execution, waiting for container requests to be fulfilled or for tasks to terminate. In the former case, the Workflow Driver requests the Workflow Scheduler to choose a task to be launched in that container. In the latter case, the Workflow Driver registers any newly produced data, which may induce new tasks becoming ready for execution and thus new container requests to be issued.

One of Hi-WAY's core strengths is its ability to interpret *iterative* workflows, which may contain unbounded loops, conditionals, and recursive tasks. In such *iterative* workflows, the termination of a task may entail the discovery of entirely new tasks. For this reason, the Workflow Driver dynamically evaluates the results of completed tasks, forwarding newly discovered tasks to the Workflow Scheduler, similar to during workflow parsing. See Figure 3 for a visualization of the Workflow Driver's execution model.

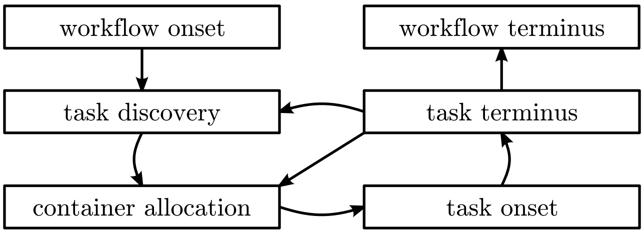


Figure 3: The *iterative* Workflow Driver's execution model. A workflow is parsed, entailing the discovery of tasks as well as the request for and eventual allocation of containers for ready tasks. Upon termination of a task executed in an allocated container, previously discovered tasks might become ready (resulting in new container requests), new tasks might be discovered, or the workflow might terminate.

As an example for an *iterative* workflow, consider an implementation of the k-means clustering algorithm commonly encountered in machine learning applications. k-means provides a heuristic for partitioning a number of data points into k clusters. To this end, over a sequence of parallelizable steps, an initial random clustering is iteratively refined until convergence is reached. Only by means of conditional task execution and unbounded iteration can this algorithm be implemented as a workflow, which underlines the importance of such *iterative* control structures in scientific workflows. The implementation of the k-means algorithm as a Cuneiform workflow has been published in [9].

3.4 Workflow Scheduler

Determining a suitable assignment of tasks to compute nodes is called workflow scheduling. To this end, the Workflow Scheduler receives tasks discovered by the Workflow Driver, from which it builds a schedule and creates container requests. Based on this schedule, the Workflow Scheduler selects a task for execution whenever a container has been allocated. This higher-level scheduler is different to YARN's internal schedulers, which, at a lower level, determine how to distribute resources between multiple users and applications. Hi-WAY provides a selection of workflow scheduling policies that optimize *performance* for different workflow structures and computational architectures.

Most established SWfMSs employ a first-come-first-served (FCFS) scheduling policy in which tasks are placed at the tail of a queue, from whose head they are removed and dispatched for execution whenever new resources become available. While Hi-WAY supports FCFS scheduling as well, its default scheduling policy is a data-aware scheduler intended for I/O-intensive workflows. The data-aware scheduler minimizes data transfer by assigning tasks to compute nodes based on the amount of input data that is already present locally. To this end, whenever a new container is allocated, the data-aware scheduler skims through all tasks pending execution, from which it selects the task with the highest fraction of input data available locally (in HDFS) on the compute node hosting the newly allocated container.

In contrast to data-aware and FCFS scheduling, static scheduling policies employ a pre-built schedule, which dictates how the tasks composing a workflow are to be assigned to available compute nodes. When configured to employ a static scheduling policy, Hi-WAY's Workflow Scheduler assembles this schedule at the beginning of workflow execution and enforces containers to be placed on specific compute nodes according to this schedule. A basic static scheduling policy supported by Hi-WAY is a round-robin scheduler that assigns tasks in turn, and thus in equal numbers, to the available compute nodes.

In addition to these scheduling policies, Hi-WAY is also able to employ *adaptive* scheduling in which the assignment of tasks to compute nodes is based on continually updated runtime estimates and is therefore adapted to the computational infrastructure. To determine such runtime estimates, the Provenance Manager, which is responsible for gathering, storing, and providing provenance and statistics data (see Section 3.5), supplies the Workflow Scheduler with exhaustive statistics. For instance, when deciding whether to assign a task to a newly allocated container on a certain compute node, the Workflow Scheduler can query the Provenance Manager for (i) the observed runtimes of earlier tasks of the same signature (i.e., invoking the same tools) running on either the same or other compute nodes, (ii) the names and sizes of the files being processed in these tasks, and (iii) the data transfer times for obtaining this input data.

If available, based on this information the Workflow Scheduler is able to determine runtime estimates for running any task on any machine. In order to quickly adapt to performance changes in the computational infrastructure, the current strategy for computing these runtime estimates is to always use the latest observed runtime. If no runtimes have been observed yet for a particular task-machine-assignment, a default runtime of zero is assumed to encourage trying out new assignments and thus obtain a more complete picture of which task performs well on which machine.

To make use of these runtime estimates, Hi-WAY supports heterogeneous earliest finish time (HEFT) [39] scheduling. HEFT exploits heterogeneity in both the tasks to be executed as well as the underlying computational infrastructure. To this end, it uses runtime estimates to rank tasks by the expected time required from task onset to workflow terminus. By decreasing rank, tasks are assigned to compute nodes with a favorable runtime estimate, i.e., critical tasks with a longer time to finish are placed on the best-performing nodes first.

Since static schedulers like round-robin and HEFT require the complete invocation graph of a workflow to be deductible at the onset of computation, static scheduling can not be used in conjunction with workflow languages that allow *iterative* workflows. Hence, the latter two (static) scheduling policies are not compatible with Cuneiform workflows (see Section 3.3).

Additional (non-static) adaptive scheduling policies are in the process of being integrated and will be described and evaluated in a separate manuscript. However, note that due to the black-box operator model, scheduling policies may not conduct structural alterations to the workflow automatically, as commonly found in database query optimization.

3.5 Provenance Manager

The Provenance Manager surveys workflow execution and registers events at different levels of granularity. First, it traces events at the workflow level, including the name of the workflow and its total execution time. Secondly, it logs events for each task, e.g., the commands invoked to spawn the task, its makespan, its standard output and error channels and the compute node on which it ran. Thirdly, it stores events for each file consumed and produced by a task. This includes its size and the time it took to move the file between HDFS and the local file system. All of this provenance data is supplemented with timestamps as well as unique identifiers and stored as JSON objects in a trace file in HDFS, from where it can be accessed by other instances of Hi-WAY.

Since this trace file holds information about all of a workflow's tasks and data dependencies, it can be interpreted as a workflow itself. Hi-WAY promotes *reproducibility* of experiments by being able to parse and execute such workflow traces directly through its Workflow Driver, albeit not necessarily on the same compute nodes. Hence, workflow trace files generated by Hi-WAY constitute a fourth supported workflow language.

Evidently, the amount of workflow traces can become difficult to handle for heavily-used installations of Hi-WAY with thousands of trace files or more. To cope with such high volumes of data, Hi-WAY provides prototypical implementations for storing and accessing this provenance data in a MySQL or Couchbase database as an alternative to storing trace files in HDFS. The usage of a database for storing this provenance data brings the added benefit of facilitating manual queries and aggregation.

3.6 Reproducible Installation

The properties of the scientific workflow programming model with its black-box data and operator models, as well as the usage of Hadoop for resource management and data distribution, both dictate requirements for workflow designers (for an illustration of some of these requirements, refer to Figure 2). First, all of a workflow's software dependencies (executables, software libraries, etc.) have to be available on each of the compute nodes managed by YARN, since any of the tasks composing a workflow could be assigned to any compute node. Secondly, any input data required to run the workflow has to be placed in HDFS or made locally available on all nodes.

To set up an installation of Hi-WAY and Hadoop, configuration routines are available online in the form of Chef recipes. Chef is a configuration management software for the automated setup of computational infrastructures [2]. These Chef installation routines, called recipes, allow for the setup of standalone or distributed Hi-WAY installations, either on local machines or in public compute clouds such as Amazon's EC2. In addition, recipes are available for setting up a large variety of execution-ready workflows. This includes obtaining their input data, placing it in HDFS, and installing any software dependencies required to run the workflow. Besides providing a broad array of use cases, these recipes enable *reproducibility* of all the experiments outlined in Section 4. The procedure of running these Chef recipes via the orchestration engine Karamel [1] to set up a distributed Hi-WAY execution environment along with a selection of workflows is described in [9] and on http://saasfee.io.

Note that this means of providing *reproducibility* exists in addition to the executable provenance traces described in Section 3.5. However, while the Chef recipes are well-suited for reproducing experiments across different research groups and compute clusters, the executable trace files are intended for use on the same cluster, since running a trace file requires input data to be located and software requirements to be available just like during the workflow run from which the trace file was derived.

4. EVALUATION

We conducted a number of experiments in which we evaluated Hi-WAY's core features of *scalability*, *performant* execution, and *adaptive* workflow scheduling. The remaining properties (support for *multilingualism*, *reproducible* experiments, and *iterative* workflows) are achieved by design. The workflows outlined in this section are written in three different languages and can be automatically set up (including input data) and run on Hi-WAY with only a few clicks following the procedure described in Section 3.6.

Across the experiments described here, we executed relevant workflows from different areas of research on both virtual clusters of Amazon's EC2 and local computational infrastructure. Section 4.1 outlines two experiments in which we analyze the *scalability* and *performance* behavior of Hi-WAY when increasing the number of available computational nodes to very large numbers. In Section 4.2, we then describe an experiment that contrasts the *performance* of running a computationally intensive Galaxy workflow on both Hi-WAY and Galaxy CloudMan. Finally, in Section 4.3 we report on an experiment in which the effect of provenance data on *adaptive* scheduling is evaluated. Table 1 gives an overview of all experiments described in this section.

4.1 Scalability / Genomics

For evaluating the *scalability* of Hi-WAY, we employed a single nucleotide variant calling workflow [31], which determines and characterizes genomic variants in a number of genomes. The input of this workflow are genomic reads emitted from a next-generation sequencing machine, which are aligned against a reference genome in the first step of the workflow using Bowtie 2 [24]. In the second step of the workflow, alignments are sorted using SAMtools [25] and genomic variants are determined using VarScan [22]. Finally, detected variants are annotated using the ANNOVAR [44] toolkit. Input data, in the form of genomic reads, was obtained from the 1000 Genomes Project [37].

In a first experiment we implemented this workflow in both Cuneiform and Tez. We ran both Hi-WAY and Tez on a Hadoop installation set up on a local cluster comprising

Table 1: Overview of conducted experiments, their evaluation goals and the section in which they are outlined.

workflow	domain	language	scheduler	infrastructure	runs	evaluation	section
SNV Calling	genomics	Cuneiform	data-aware	24 Xeon E5-2620	3	*performance*, *scalability*	4.1
SNV Calling	genomics	Cuneiform	FCFS	128 EC2 m3.large	3	*scalability*	4.1
RNA-seq	bioinformatics	Galaxy	data-aware	6 EC2 c3.2xlarge	5	*performance*	4.2
Montage	astronomy	DAX	HEFT	8 EC2 m3.large	80	*adaptive* scheduling	4.3

24 compute nodes connected via a one gigabit switch. Each compute node provided 24 gigabyte of memory as well as two Intel Xeon E5-2620 processors with 24 virtual cores. This resulted in a maximum of 576 concurrently running containers, of which each one was provided with its own virtual processor core and one gigabyte of memory.

The results of this experiment are illustrated in Figure 4. *Scalability* beyond 96 containers was limited by network bandwidth. The results indicate that Hi-WAY *performs* comparably to Tez while network resources are sufficient, yet *scales* favorably in light of limited network resources due to its data-aware scheduling policy, which reduced data transfer by preferring to assign the data-intensive reference alignment tasks to containers on compute nodes with a locally available replicate of the input data. However, probably the most important finding of this experiment was that the implementation of the workflow in Cuneiform resulted in very little code and was finished in a few days, whereas it took several weeks and a lot of code in Tez.

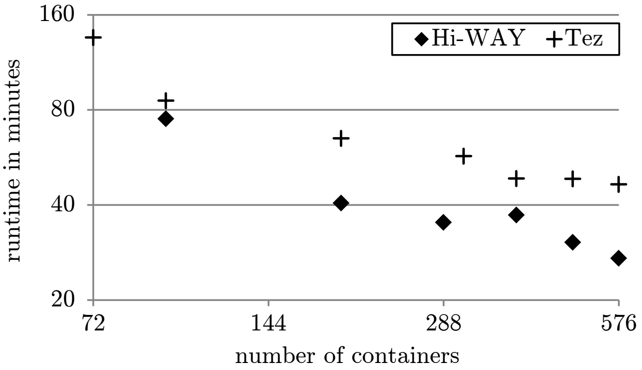


Figure 4: Mean runtimes of the variant calling workflow with increasing number of containers. Note that both axes are in logarithmic scale.

In a second experiment, we increased the volume of input data while at the same time reducing network load by (i) using additional genomic read files from the 1000 Genomes Project, (ii) compressing intermediate alignment data using CRAM referential compression [25], and (iii) obtaining input read data during workflow execution from the Amazon S3 bucket of the 1000 Genomes Project instead of storing them on the cluster in HDFS.

In the process of this second experiment, the workflow was first run using a single worker node, processing a single genomic sample comprising eight files, each about one gigabyte in size, thus amounting to eight gigabytes of input data in total. In subsequent runs, we then repeatedly doubled the number of worker nodes and volume of input data. In the last run (after seven duplications), the computational infrastructure consisted of 128 worker nodes, whereas the workflow's input data comprised 128 samples of eight roughly gigabyte-sized files each, amounting to a total volume of more than a terabyte of data.

The experiment was run three times on virtual clusters of Amazon's EC2. To investigate potential effects of datacenter locality on workflow runtime (which we did not observe during the experiment), these clusters were set up in different EC2 regions – once in the EU West (Ireland) and twice in the US East (North Virginia) region. Since we intended to analyze the *scalability* of Hi-WAY, we isolated the Hi-WAY AM from the worker threads and Hadoop's master threads. To this end, dedicated compute nodes were provided for (i) the Hi-WAY AM, running in its own YARN container, and (ii) the two Hadoop master threads (HDFS's NameNode and YARN's ResourceManager). All compute nodes – the two master nodes and all of the up to 128 worker nodes – were configured to be of type m3.large, each providing two virtual processing cores, 7.5 gigabytes of main memory, and 32 gigabytes of local SSD storage.

All of the experiment runs were set up automatically using Karamel [1]. Over the course of the experiment we determined the runtime of the workflow. Furthermore, the CPU, I/O, and network performance of the master and worker nodes was monitored during workflow execution using the Linux tools *uptime*, *ifstat*, and *iostat*. Since the workflow's tasks required the whole memory provided by a single compute node, we configured Hi-WAY to only allow a single container per worker node at the same time, enabling multithreading for tasks running within that container whenever possible. Hi-WAY was configured to utilize the basic FCFS queue scheduler (see Section 3.4). Other than that, both Hi-WAY and Hadoop were set up with default parameters.

The average of measured runtimes with steadily increasing amounts of both compute nodes and input data is displayed in Table 2 and Figure 5. The regression curve indicates near-linear *scalability*: The doubling of input data and the associated doubling of workload is almost fully offset by a doubling of worker nodes. This is even true for the maximum investigated cluster size of 128 nodes, in which a terabyte of genomic reads was aligned and analyzed against the whole human genome. Note that extrapolating the average runtime for processing eight gigabytes of data on a single machine reveals that aligning a whole terabyte of genomic read data against the whole human genome along with further downstream processing would easily take a month on a single machine.

We identified and evaluated several potential bottlenecks when scaling out a Hi-WAY installation beyond 128 nodes. For instance, Hadoop's master processes, YARN's ResourceManager and HDFS's NameNode, could prove to limit *scalability*. Similarly, the Hi-WAY AM process that handles the scheduling of tasks, the assembly of results, and the tracing of provenance, could collapse when further increasing the workload and the number of available compute nodes. To

Table 2: Summary of the scalability experiment described in Section 4.1. The number of provisioned VMs is displayed alongside the volume of processed data, average runtime (over three runs), and the incurred cost.

number of worker VMs	1	2	4	8	16	32	64	128
number of master VMs	2	2	2	2	2	2	2	2
data volume	8.06 GB	16.97 GB	33.10 GB	69.47 GB	136.14 GB	270.98 GB	546.76 GB	1096.83 GB
avg. runtime in min.	340.12	350.36	351.62	344.82	375.57	372.09	380.24	353.39
runtime std. dev.	1.96	0.14	2.15	1.88	14.84	22.10	22.34	6.01
avg. cost[1] per run	$2.48	$3.41	$5.13	$8.39	$16.45	$30.78	$61.07	$111.79
avg. cost[1] per GB	$0.31	$0.20	$0.16	$0.12	$0.12	$0.11	$0.11	$0.10

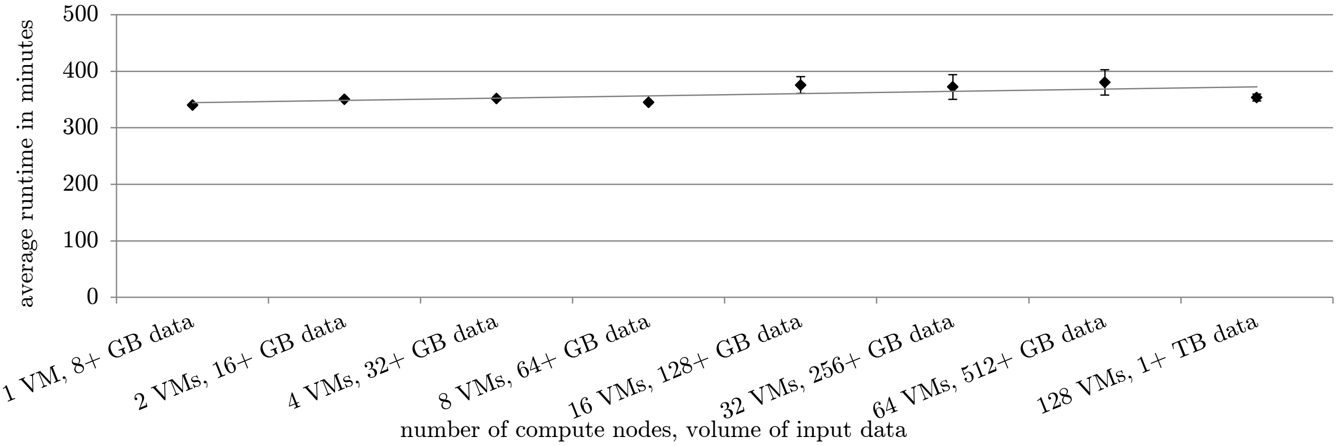


Figure 5: Mean runtimes for three runs of the variant calling workflow described in Section 4.1 when repeatedly doubling the number of compute nodes available to Hi-WAY along with the input data to be processed. The error bars represent the standard deviation, whereas the line represents the (linear) regression curve[2].

this end, we were interested in the resource utilization of these potential bottlenecks, which is displayed in Figure 6.

We observe a steady increase in load across all resources for the Hadoop and Hi-WAY master nodes when repeatedly doubling the workload and number of worker nodes. However, resource load stays well below maximum utilization at all cluster sizes. In fact, all resources are still utilized less than 5 % even when processing one terabyte of data across 128 worker nodes. Furthermore, we observe that resource utilization for Hi-WAY's master process is of the same order of magnitude as for Hadoop's master processes, which have been developed to scale to 10,000 compute nodes and beyond [42].

While resource utilization on the master nodes increases when growing the workload and computational infrastructure, we observe that CPU utilization stays close to the maximum of 2.0 on the worker nodes, whereas the other resources stay under-utilized. This finding is unsurprising, since both the alignment step and the variant calling step of the workflow support multithreading and are known to be CPU-bound. Hence, this finding confirms that the cluster is nearly fully utilized for processing the workflow, whereas the master processes appear to be able to cope with a considerable amount of additional load.

4.2 Performance / Bioinformatics

RNA sequencing (RNA-seq) methodology makes use of next-generation sequencing technology to enable researchers to determine and quantify the transcription of genes in a given tissue sample. Trapnell et al. [40] have developed a workflow that has has been established as the *de facto* standard for processing and comparing RNA-seq data.

In the first step of this workflow, genomic reads are aligned against a reference genome using the two alignment tools Bowtie 2 [24] and TopHat 2 [21]. The alignment serves the purpose of identifying the reads' genomic positions, which have been lost during the sequencing process. This first step is comparable to the first step in the variant calling workflow described in Section 4.1. However, in this workflow, reads are obtained by sequencing only the transcriptome, i.e., the set of transcribed genes, as opposed to sequencing the whole genome. In the second step of the workflow, the Cufflinks [40] package is utilized to assemble and quantify transcripts of genes from these aligned reads and, finally, to compare quantified transcripts for different input samples, for instance between diseased and healthy samples. See Figure 7 for a visualization of the RNA-seq workflow.

[1] Here, we assume a price of $0.146 per minute, as listed for m3.large instances in EC2's EU West region at the time of writing. We also assume billing per minute and disregard time required to set up the experiment.

[2] The standard deviation is higher for cluster sizes of 16, 32, and 64 nodes, which is due to the observed runtime of the CPU-bound variant calling step being notably higher in one run of the experiment. Since these three measurements were temporally co-located and we did not observe similar distortions at any other point in time, this observation can most likely be attributed to external factors.

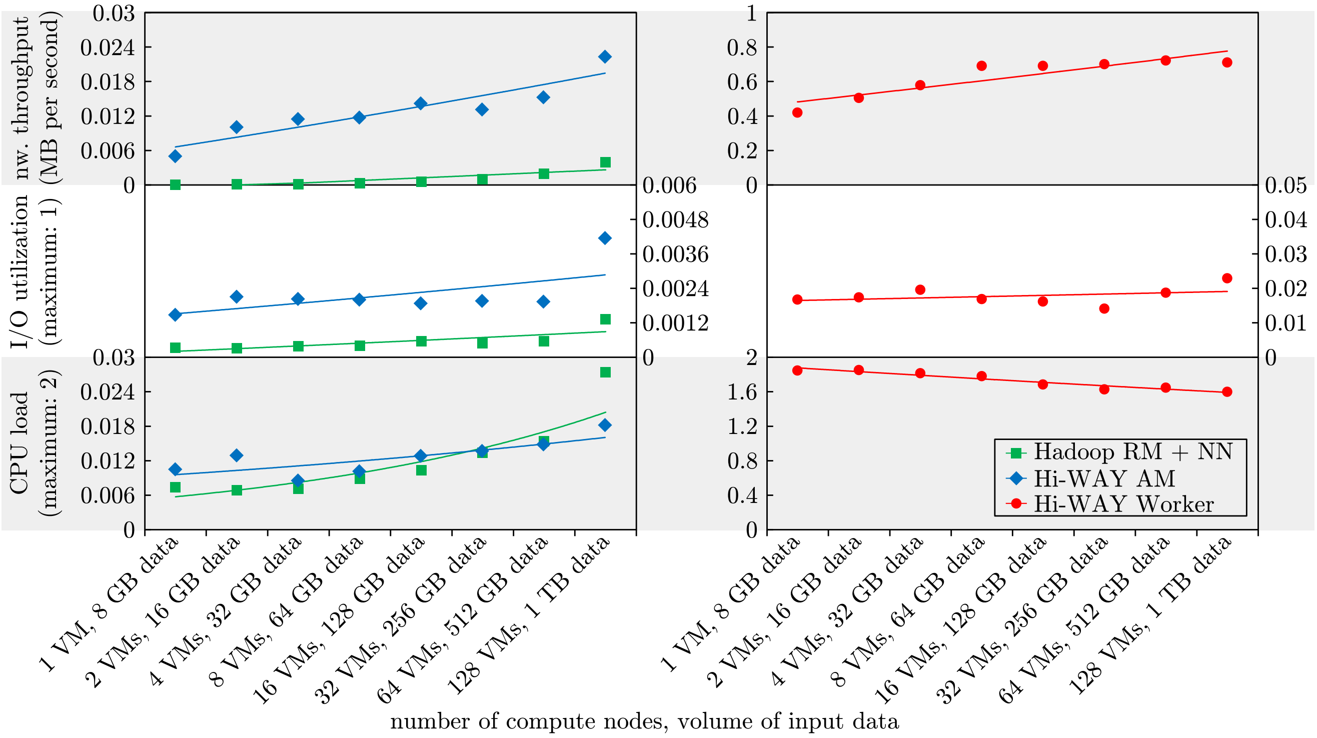


Figure 6: Resource utilization (CPU load, I/O utilization, and network throughput) of virtual machines hosting the Hadoop master processes, the Hi-WAY AM and a Hi-WAY worker process. Average values over the time of workflow execution and across experiment runs are shown along with their exponential regression curve. We observed the following peak values for worker nodes: 2.0 for CPU load (due to two virtual processing cores being available per machine), 1.0 for I/O utilization (since 1.0 corresponds to device saturation, i.e., 100 % of CPU time spent for I/O requests) and 109.35 MB per second for network throughput. Note the different scales for the master nodes on the left and the worker nodes on the right.

Wolfien et al. [46] implemented an extended version of this workflow in Galaxy, making it available through Galaxy's public workflow repository. Their implementation of the workflow, called TRAPLINE, compares two genomic samples. Since each of these two samples is expected to be available in triplicates and the majority of data processing tasks composing the workflow are arranged in sequential order, the workflow, without any manual alterations, has a degree of parallelism of six across most of its parts.

We executed the TRAPLINE workflow on virtual clusters of Amazon's EC2 consisting of compute nodes of type c3.2xlarge. Each of these nodes provides eight virtual processing cores, 15 gigabytes of main memory and 160 gigabytes of local SSD storage. Due to the workflow's degree of parallelism of six, we ran the workflow on clusters of sizes one up to six. For each cluster size, we executed this Galaxy workflow five times on Hi-WAY, comparing the average runtime against an execution on Galaxy CloudMan. Each run was launched in its own cluster, set up in Amazon's US East (North Virginia) region.

As workflow input data, we used RNA-seq data of young versus aged mice, obtained from the Gene Expression Omnibus (GEO) repository[3], amounting to more than ten gigabytes in total. We set up Hi-WAY using Karamel [1] and CloudMan using its Cloud Launch web application. Default parameters were left unchanged. However, since several tasks in TRAPLINE require large amounts of memory, we configured both Hi-WAY as well as CloudMan's default underlying distributed resource manager, Slurm, to only allow execution of a single task per worker node at any time. Omitting this configuration would lead either of the two systems to run out of memory at some point during workflow execution. The results of executing the TRAPLINE workflow on both Hi-WAY and Galaxy CloudMan are displayed in Figure 8. Across all of the tested cluster sizes, we observed that Hi-WAY *outperformed* Galaxy CloudMan by at least 25 %. These differences were found to be significant by means of a one-sample *t*-test (*p*-values of 0.000127 and lower).

The observed difference in *performance* is most notable in the computationally costly TopHat2 step, which makes heavy use of multithreading and generates large amounts of intermediate files. Therefore, this finding can be attributed to Hi-WAY utilizing the worker node's transient local SSD storage, since both HDFS as well as the storage of YARN containers reside on the local file system. Conversely, Galaxy CloudMan stores all of its data on an Amazon Elastic Block Store (EBS) volume, a persistent drive that is accessed over the network and shared among all compute nodes[4].

[3] series GSE62762, samples GSM15330[14|15|16|45|46|47]

[4] While EBS continues to be CloudMan's default storage option, a recent update has introduced support for using transient storage instead.

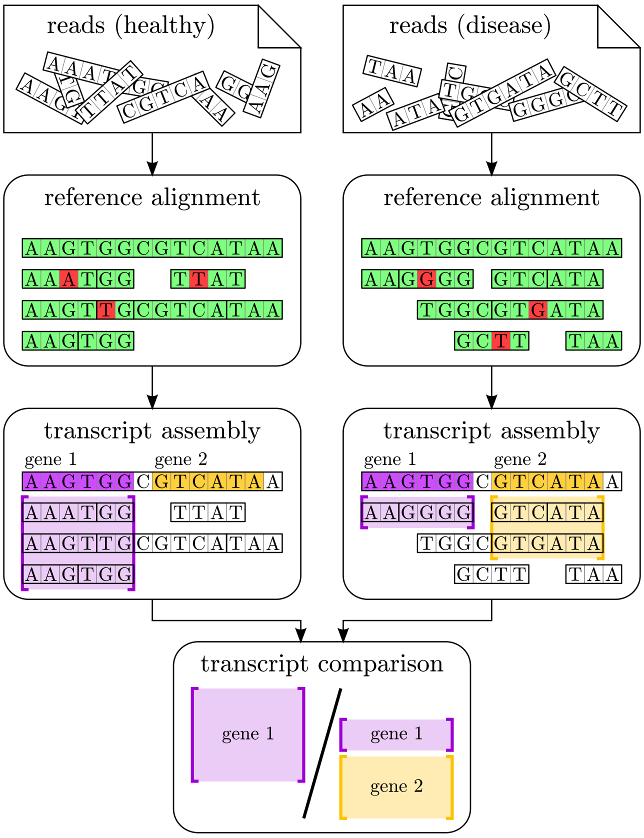


Figure 7: The RNA sequencing workflow described in Section 4.2. Genomic reads, the output of whole transcriptome sequencing, are aligned against a reference genome. Transcribed genes are then determined and quantified based on these alignments. Finally, transcription is compared between samples.

Apart from the observed gap in *performance*, it is important to point out that Galaxy CloudMan only supports the automated setup of virtual clusters of up to 20 nodes. Compared to Hi-WAY, it therefore only provides very limited *scalability*. We conclude that Hi-WAY leverages the strengths of Galaxy, which lie in its intuitive means of workflow design and vast number of supported tools, by providing a more *performant*, flexible, and *scalable* alternative to Galaxy CloudMan for executing data-intensive Galaxy workflows with a high degree of parallelism.

4.3 Adaptive Scheduling / Astronomy

To underline the benefits of *adaptive* scheduling on heterogeneous computational infrastructures, an additional experiment was performed in which we generated a Pegasus DAX workflow using the Montage toolkit [7]. The resulting workflow assembles a 0.25 degree mosaic image of the Omega Nebula. It comprises a number of steps in which images obtained from telescopic readings are projected onto a common plane, analyzed for overlapping regions, cleaned from background radiation noise and finally merged into a mosaic.

The Montage toolkit can be used to generate workflows with very large number of tasks by increasing the degree value. However, the degree of 0.25 used in this experiment resulted in a comparably small workflow with a maximum degree of parallelism of eleven during the image projection and background radiation correction phases of the workflow. In the experiment, this workflow was repeatedly executed on a Hi-WAY installation set up on a virtual cluster in the EU West (Ireland) region of Amazon's EC2. The cluster comprised a single master node as well as eleven worker nodes to match the workflow's degree of parallelism. Similar to the *scalability* experiment in Section 4.1, all of the provisioned virtual machines were of type m3.large.

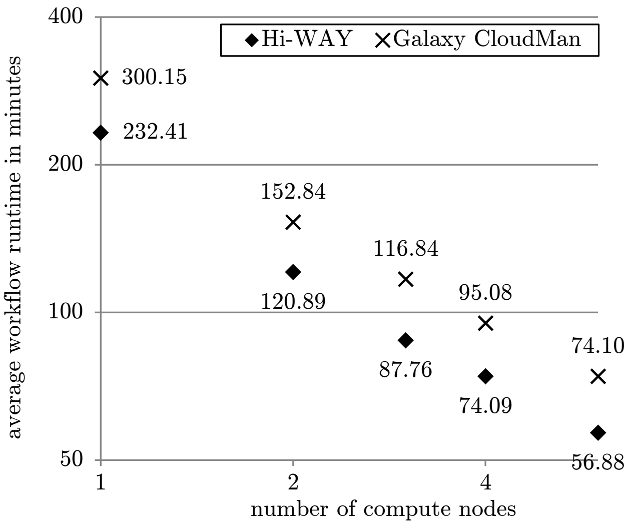


Figure 8: Average runtime of executing the RNA-seq workflow described in Section 4.2 on Hi-WAY and Galaxy CloudMan. The number of EC2 compute nodes of type c3.2xlarge was increased from one up to six. Note that both axes are in logarithmic scale.

To simulate a heterogeneous and potentially shared computational infrastructure, synthetic load was introduced on these machines by means of the Linux tool *stress*. To this end, only one worker machine was left unperturbed, whereas five worker machines were taxed with increasingly many CPU-bound processes and five other machines were impaired by launching increasingly many (in both cases 1, 4, 16, 64, and 256) processes writing data to the local disk.

A single run of the experiment, of which 80 were conducted in total, encompassed (i) running the Montage workflow once using a FCFS scheduling policy, which served as a baseline to compare against, and (ii) running the workflow 20 times consecutively using the HEFT scheduler. In the process of these consecutive runs, larger and larger amounts of provenance data became available over time as a consequence of prior workflow executions. Hence, workflow executions using the HEFT scheduler were provided with increasingly comprehensive runtime estimates. Between iterations however, all provenance data was removed.

The results of this experiment are illustrated in Figure 9. Evidently, the performance of HEFT scheduling improves with more and more provenance data becoming available. Employing HEFT scheduling in the absence of any available provenance data results in subpar performance compared to FCFS scheduling. This is due to HEFT being a static scheduling policy, which entails that task assignments are fixed, even if one worker node still has many tasks to run while another, possibly more performant worker node is idle.

However, with a single prior workflow run, HEFT already outperforms FCFS scheduling significantly (two-sample t-test, p-value of 0.033). The next significant performance gain can then be observed between ten and eleven prior workflow execution (two-sample t-test, p-value of $6.22 \cdot 10^{-7}$). At this point, any task composing the workflow, even the ones that are only executed once per workflow run, have been executed on all eleven worker nodes at least once. Hence, runtime estimates are complete and scheduling is no longer driven by the need to test additional task-machine-assignments. Note that this also leads to more stable workflow runtimes, which is reflected in a major reduction of the standard deviation of runtime. We argue that the observed performance gains of HEFT over baseline FCFS scheduling emphasize the importance of and potential for *adaptive* scheduling in distributed scientific workflow execution.

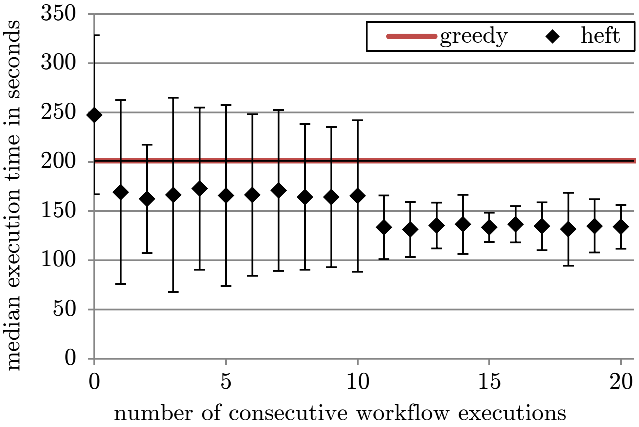


Figure 9: Median runtime of executing Montage on a heterogeneous infrastructure when using HEFT scheduling and increasing the number of previous workflow runs and thus the amount of available provenance data. The error bars represent the standard deviation.

5. CONCLUSION AND FUTURE WORK

In this paper, we presented Hi-WAY, an application master for executing arbitrary scientific workflows on top of Hadoop YARN. Hi-WAY's core features are a *multilingual* workflow language interface, support for *iterative* workflow structures, *adaptive* scheduling policies optimizing *performance*, tools to provide *reproducibility* of experiments, and, by employing Hadoop for resource management and storage, *scalability*. We described Hi-WAY's interface with YARN as well as its architecture, which is built around the aforementioned concepts. We then outlined four experiments, in which real-life workflows from different domains were executed on different computational infrastructures comprising up to 128 worker machines.

As future work, we intend to further harness the statistics on resource utilization provided by Hi-WAY's Provenance Manager. Currently, the containers requested by Hi-WAY and provided by YARN all share an identical configuration, i.e., they all have the same amounts of virtual processing cores and memory. This can lead to under-utilization of resources, since some tasks might not be able to put all of the provided resources to use. To this end, we intend to extend Hi-WAY with a mode of operation, in which containers are custom-tailored to the tasks that are to be executed.

Funding

Marc Bux and Jörgen Brandt were funded by the EU project BiobankCloud. We also acknowledge funding by the DFG graduate school SOAMED and support by an AWS in Education Grant award.

References

[1] Karamel. http://www.karamel.io.

[2] Opscode Chef. https://www.chef.io.

[3] E. Afgan et al. Galaxy CloudMan: Delivering Cloud Compute Clusters. *BMC Bioinformatics*, 11(Suppl 12):S4, 2010.

[4] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids. *SWEET Workshop*, 2012.

[5] A. Alexandrov et al. The Stratosphere Platform for Big Data Analytics. *VLDBJ*, 23(6):939–964, 2014.

[6] P. Amstutz et al. Common Workflow Language, v1.0. *Figshare*, 2016.

[7] G. B. Berriman et al. Montage: A Grid-Enabled Engine for Delivering Custom Science-Grade Mosaics on Demand. *SPIE Astronomical Telescopes + Instrumentation*, 5493:221–232, 2004.

[8] J. Brandt, M. Bux, and U. Leser. Cuneiform: A Functional Language for Large Scale Scientific Data Analysis. *Workshops of the EDBT/ICDT*, 1330:17–26, 2015.

[9] M. Bux et al. Saasfee: Scalable Scientific Workflow Execution Engine. *PVLDB*, 8(12):1892–1895, 2015.

[10] M. Bux and U. Leser. Parallelization in Scientific Workflow Management Systems. *arXiv preprint, arXiv:1303.7195*, 2013.

[11] S. Cohen-Boulakia and U. Leser. Search, Adapt, and Reuse: The Future of Scientific Workflows. *SIGMOD Record*, 40(2):6–16, 2011.

[12] S. B. Davidson and J. Freire. Provenance and Scientific Workflows: Challenges and Opportunities. *SIGMOD Conference*, 2008.

[13] E. Deelman et al. Pegasus, a Workflow Management System for Large-Scale Science. *Future Generation Computer Systems*, 46:17–35, 2015.

[14] P. Di Tommaso et al. The Impact of Docker Containers on the Performance of Genomic Pipelines. *PeerJ*, 3:e1273, 2015.

[15] Y. Gil et al. Examining the Challenges of Scientific Workflows. *Computer*, 40(12):26–34, 2007.

[16] C. Goble and D. de Roure. myExperiment: Social Networking for Workflow-Using e-Scientists. *WORKS Workshop*, 2007.

[17] J. Goecks, A. Nekrutenko, and J. Taylor. Galaxy: a Comprehensive Approach for Supporting Accessible, Reproducible, and Transparent Computational Research in the Life Sciences. *Genome Biology*, 11(8):R86, 2010.

[18] T. Hey, S. Tansley, and K. Tolle. *The Fourth Paradigm*. Microsoft Research, 2nd edition, 2009.

[19] B. Hindman et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. *NSDI Conference*, 2011.

[20] M. Islam et al. Oozie: Towards a Scalable Workflow Management System for Hadoop. *SWEET Workshop*, 2012.

[21] D. Kim et al. TopHat2: Accurate Alignment of Transcriptomes in the Presence of Insertions, Deletions and Gene Fusions. *Genome Biology*, 14:R36, 2013.

[22] D. C. Koboldt et al. VarScan: Variant Detection in Massively Parallel Sequencing of Individual and Pooled Samples. *Bioinformatics*, 25(17):2283–2285, 2009.

[23] J. Köster and S. Rahmann. Snakemake – a Scalable Bioinformatics Workflow Engine. *Bioinformatics*, 28(19):2520–2522, 2012.

[24] B. Langmead and S. Salzberg. Fast Gapped-Read Alignment with Bowtie 2. *Nature Methods*, 9:357–359, 2012.

[25] H. Li et al. The Sequence Alignment/Map Format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[26] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso. A Survey of Data-Intensive Scientific Workflow Management. *Journal of Grid Computing*, 13(4):457–493, 2015.

[27] I. Momcheva and E. Tollerud. Software Use in Astronomy: An Informal Survey. *arXiv preprint, arXiv:1507.03989*, 2015.

[28] D. G. Murray et al. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. *NSDI Conference*, 2011.

[29] E. Ogasawara et al. An Algebraic Approach for Data-Centric Scientific Workflows. In *PVLDB*, volume 4, pages 1328–1339, 2011.

[30] E. Ogasawara et al. Chiron: A Parallel Engine for Algebraic Scientific Workflows. *Concurrency and Computation: Practice and Experience*, 25(16):2327–2341, 2013.

[31] S. Pabinger et al. A Survey of Tools for Variant Analysis of Next-Generation Genome Sequencing Data. *Briefings in Bioinformatics*, 15(2):256–278, 2014.

[32] B. Saha et al. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. *SIGMOD Conference*, 2015.

[33] I. Santana-Perez et al. Leveraging Semantics to Improve Reproducibility in Scientific Workflows. *Reproducibility at XSEDE Workshop*, 2014.

[34] Z. D. Stephens et al. Big Data: Astronomical or Genomical? *PLoS Biology*, 13(7):e1002195, 2015.

[35] R. Sumbaly, J. Kreps, and S. Shah. The Big Data Ecosystem at LinkedIn. *SIGMOD Conference*, 2013.

[36] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields. *Workflows for e-Science: Scientific Workflows for Grids*. Springer, 2007.

[37] The 1000 Genomes Project Consortium. A Global Reference for Human Genetic Variation. *Nature*, 526(7571):68–74, 2015.

[38] P. D. Tommaso, M. Chatzou, P. P. Baraja, and C. Notredame. A Novel Tool for Highly Scalable Computational Pipelines. *Figshare*, 2014.

[39] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[40] C. Trapnell et al. Differential Gene and Transcript Expression Analysis of RNA-seq Experiments with TopHat and Cufflinks. *Nature Protocols*, 7(3):562–578, 2012.

[41] E. L. Van Dijk, H. Auger, Y. Jaszczyszyn, and C. Thermes. Ten Years of Next-Generation Sequencing Technology. *Trends in Genetics*, 30(9):418–426, 2014.

[42] V. K. Vavilapalli et al. Apache Hadoop YARN: Yet Another Resource Negotiator. *ACM SOCC*, 2013.

[43] J. Vivian et al. Rapid and Efficient Analysis of 20,000 RNA-seq Samples with Toil. *bioRxiv*, 062497, 2016.

[44] K. Wang, M. Li, and H. Hakonarson. ANNOVAR: Functional Annotation of Genetic Variants from High-Throughput Sequencing Data. *Nucleic Acids Research*, 38(16):e164, 2010.

[45] M. Wilde et al. Swift: A Language for Distributed Parallel Scripting. *Parallel Computing*, 37(9):633–652, 2011.

[46] M. Wolfien et al. TRAPLINE: A Standardized and Automated Pipeline for RNA Sequencing Data Analysis, Evaluation and Annotation. *BMC Bioinformatics*, 17:21, 2016.

[47] K. Wolstencroft et al. The Taverna Workflow Suite: Designing and Executing Workflows of Web Services on the Desktop, Web or in the Cloud. *Nucleic Acids Research*, 41:557–561, 2013.

[48] D. Wu et al. Building Pipelines for Heterogeneous Execution Environments for Big Data Processing. *Software*, 33(2):60–67, 2016.

[49] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. *HotCloud Conference*, 2010.

[50] Y. Zhao et al. Enabling scalable scientific workflow management in the Cloud. *Future Generation Computer Systems*, 46:3–16, 2015.

Buddy Instance - A Mechanism for Increasing Availability in Shared-Disk Clusters

Anjan Kumar Amirishetty, Yunrui Li, Tolga Yurek, Mahesh Girkar, Wilson Chan, Graham Ivey, Vsevolod Panteleenko, Ken Wong

Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065, U.S.A.

{Anjan.Kumar.Amirishetty}@oracle.com

ABSTRACT

Oracle's Real Application Cluster (RAC) allows multiple database instances to run on different server nodes in a cluster against a shared set of data files. A critical aspect of an Oracle RAC system is that of *instance recovery*. When a node suffers from a hardware failure, or a database instance suffers from a software failure, instance recovery is performed by a surviving instance to ensure that the database remains in a consistent state. High-availability comes from the surviving database instances, each running on a surviving node, that are still able to provide database services. During instance recovery, the set of database resources that are in need of recovery must be identified and then repaired. Until such time as the identification of these resources has been done, Oracle needs to block any requests by database clients to all database resources. The whole database appears to be frozen during this time, a period that is called application brown-out. In the interests of availability it is therefore important that instance recovery endeavors to keep this period of identification as short as possible. In doing so, not only is the brown-out period reduced, but also the overall time to make available those resources that need repair, is reduced.

This paper describes the use of a *Buddy Instance*, a mechanism that significantly reduces the brown-out time and therefore also, the duration of instance recovery. Each database instance has a buddy database instance whose purpose is to construct in-memory metadata that describes the resources needing recovery, on a continuous basis at run-time. In the event of node or instance failure, the buddy instance for the failed instance uses the in-memory metadata in performing instance recovery. The buddy instance mechanism for single instance failures is available in the 12.2 release of Oracle Database. Performance results show a significant reduction in brown-out time and also in overall instance recovery time.

Categories and Subject Descriptors

H.2.4 **[Database Management Systems]:** Database transaction processing→ Database recovery, C.4 [**Performance of Systems**]: reliability availability and serviceability

General Terms

Algorithms, Design, Performance

Keywords

Database, Real Application Cluster, Recovery, Availability

1. INTRODUCTION

Oracle RAC [1] transparently extends database applications from single-node systems to multi-node systems which share the disks that provide storage for the database. The database spans multiple hardware systems yet appears as a single unified database to the application. An *instance* is a collection of processes and memory accessing a set of data files. Single-instance Oracle databases have a one-to-one relationship between the database and the instance. Oracle RAC environments, however, have a one-to-many relationship between the database and instances.

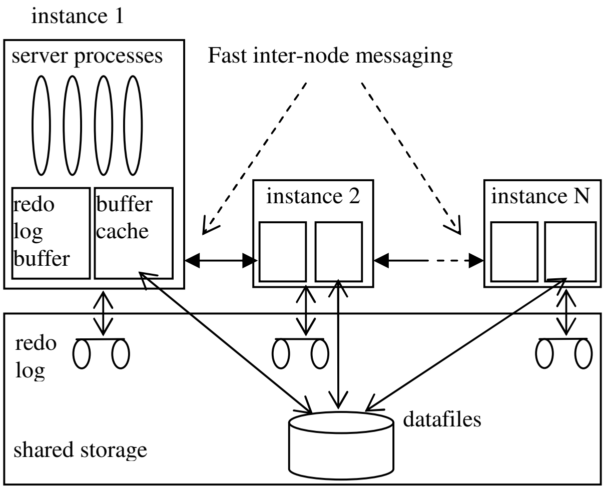


Figure 1. Real Application Cluster Architecture

An interconnect serves as the communication path between each node in the cluster database. Each Oracle instance uses the interconnect to exchange messages that synchronize each instance's use of shared resources.

RAC is so called since it transparently allows any database application to run on a cluster without requiring any application changes. RAC improves application performance since the application is executed in parallel across multiple systems. RAC also improves availability since the application is available as long as at least one of the cluster nodes is alive.

Each database instance in RAC has its own redo log. The redo log is a set of files that records all the changes to the database that have been made by the instance.

An Oracle RAC database is a shared everything database. All data files and redo log files must reside on cluster-aware shared disks so that all the instances can access these storage components.

In Oracle RAC, Cache Fusion [3] allows the data blocks to be shipped directly between Oracle instances through fast inter-node messaging, without requiring expensive disk I/O. Oracle instances therefore directly share the contents of their volatile buffer caches [8], resulting in a *shared-cache* clustered database architecture.

When some but not all instances of an Oracle RAC database fail, instance recovery is performed automatically by a surviving instance in the cluster. Instance recovery ensures that the database is in a consistent state after such a failure.

Instance recovery is done in two phases. The first phase, *cache recovery* or *rolling forward* [7], involves reapplying (or rolling forward) all necessary changes recorded in the redo log to the data blocks of data files. After cache recovery, data files could contain the changes of transactions that had not yet been committed at the time of failure.

The second phase of instance recovery, *transaction recovery* or *rolling back* [7], uses changes recorded in the undo segment to roll back uncommitted changes in data blocks. After transaction recovery, data files reflect a transactionally-consistent image of the database at the time of failure.

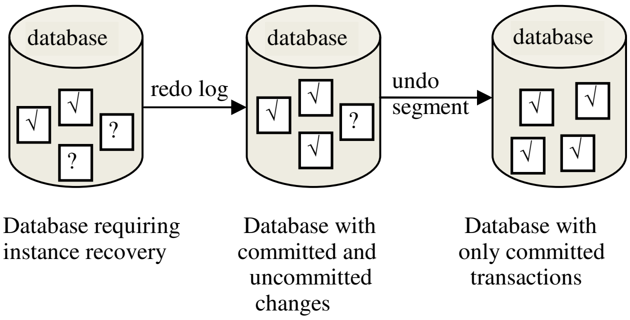

database
redo log
database
undo segment
database
√
?
Database requiring instance recovery
Database with committed and uncommitted changes
Database with only committed transactions

Figure 2. Recovery Phases: Cache Recovery and Transaction Recovery

Cache recovery must scan the redo log of each failed instance to recover the data blocks that were lost when these instances failed. Cache recovery scans the redo log in two passes [2]. The first pass constructs the metadata that is subsequently used by the second pass to speed-up recovery. Section 2 discusses the details of cache recovery in more detail.

The buddy instance mechanism potentially eliminates the first pass of cache recovery, thereby improving the performance of instance recovery.

Each instance in the cluster becomes a *protected instance* when another instance is designated to serve it as its *buddy instance*. As the protected instance records changes to the database in its redo log at runtime, its buddy instance proactively scans its log to build the metadata that the second pass can make use of if a failure were to happen at this moment. When the protected instance fails, its buddy instance performs cache recovery for the failed instance and uses the metadata it has accumulated to shortcut this process.

The rest of this paper is organized as follows. First, the motivation behind this new technique is described. After this, the existing two-pass recovery scheme is outlined. Then, the buddy instance mechanism that optimizes the two-pass recovery scheme for single instance failure is detailed. Following this there is a discussion on extending the buddy instance mechanism to multi-instance failure. Finally a performance study is tabled and related work is looked at.

2. MOTIVATING USE CASE

A major e-retail customer of the Oracle RAC database has been impacted by application brown-out that happens during instance recovery. The vast majority of these failures were single instance failures. A mechanism was needed to improve availability by reducing the length of brown-out. It was clear that this use case was not a specific one and that any improvements made, would benefit the majority of customers using RAC.

As the number of nodes increase in Oracle RAC database, probability of node failure increases and there is a need to perform instance recovery in a seamless manner, without affecting the database throughput.

The result was the buddy instance functionality, made available in the 12.2 release of Oracle Database.

3. CACHE RECOVERY

Each Oracle RAC instance is configured with its own cache of disk buffers which together, form a global buffer cache. In order to maintain cache coherency across this, global resource control is needed. The Global Cache Service (GCS) [3] tracks and maintains the locations and access modes of all data blocks in the global cache thereby maintaining the consistency of the database at the cluster level. Database blocks accessed concurrently by cluster instances have corresponding GCS resources to ensure the same data block is not updated without coordination across different instances.

GCS adopts a distributed architecture. Each instance shares the responsibility of managing a subset of the global cache. GCS maintains the status of global cache resources to ensure the overall consistency of database. When one or more instances fail, Oracle needs to rebuild the global cache resource information. Only the cache resources that reside on or are mastered by the GCS on the failed instances, need to be rebuilt or re-mastered.

3.1 Checkpointing

Cache recovery uses checkpoints to determine the set of changes that must be applied to the data files. A checkpoint represents the point at which all changes to the database have been made persistent.

Each instance in Oracle has its own redo log which is effectively, an ever-growing list of redo records generated by an instance [4]. The position of each record in the redo log may be identified by its *redo byte address* (RBA) [4]. The location of the checkpoint is identified using the checkpoint RBA. This is the position in the redo log of an instance at which all changes to data blocks made by that instance are known to be on disk. Hence, recovery for that instance needs to recover only those data blocks whose redo records occur between the checkpoint RBA and the end of the log [4].

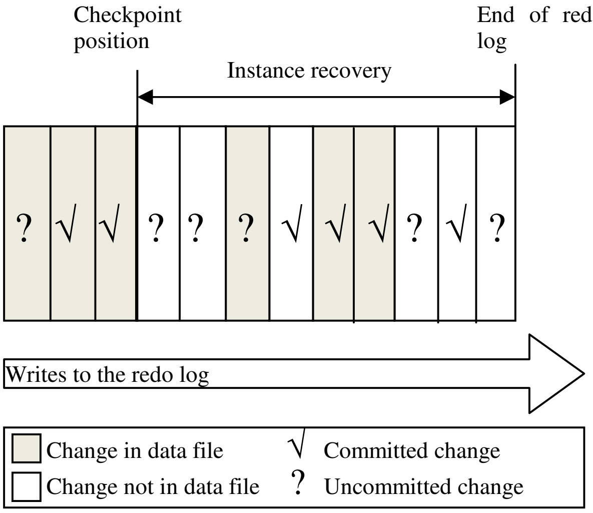


Figure 3. Checkpoint Position in Redo Log

Cache recovery must scan the redo log of each failed instance and apply the changes that occur between the marker for the last checkpoint and the end of the redo log.

3.1.1 Instance Checkpoint

The low RBA for a disk buffer is the RBA corresponding to the first (in-memory) modification of the data block. Oracle Database maintains a buffer checkpoint queue [4] which contains modified buffers linked in ascending order of their *low RBA*. Each buffer header contains the value of the low RBA associated with the buffer; this value is set when the buffer is first modified. A buffer that contains a yet-to-be-changed block does not have a low RBA in its buffer header and is not linked on the checkpoint queue. After a changed buffer is written, it is unlinked from its checkpoint queue.

As buffers from the head of the queue are written to disk, the instance checkpoint (lowest low-RBA of the modified buffers) will keep advancing [4]. This lowest low-RBA is referred to as the current position of the *instance checkpoint* for the instance. The instance checkpoint advances the database checkpoint RBA as a lightweight background activity.

3.1.2 Database Checkpoint

Each change in Oracle is associated with a time, known as the *system change number* (SCN). Instance checkpoint is also associated with a SCN. The *database checkpoint* in RAC is the instance checkpoint that has the lowest checkpoint SCN of all the instances.

3.2 Two-pass Recovery Scheme

Instance recovery for all failed instances is triggered automatically on a surviving instance. Oracle uses a two-pass database recovery scheme [2] to recover the changes to data blocks that were lost on the failed instances. The first pass scans the redo logs of each failed instance to decide the data blocks that need to be recovered. This list of blocks is referred to as the *recovery set*. The second pass applies redo from the redo logs to the blocks in the recovery set.

A *Block Written Record* (BWR) is recorded in the redo log whenever an instance writes a block to a data file. When the first pass encounters a BWR, the corresponding data block entry in the recovery set is removed because it is known that at this point in time, the block changes have been made persistent on disk. BWRs allow instance recovery to avoid unnecessary reads of data blocks that were not modified between being written to disk and the point at which instance failure occurred. BWRs ensure that the recovery set constructed by the first pass is much smaller than the total number of blocks that were actually modified on the failed instances.

The first and second passes both start at the lowest checkpoint SCN of all failed instances. The redo records of all the failed instances are merged in SCN order. In both passes, Oracle scans the redo until the end of all redo logs for all the failed instances, proceeding through as many log files as necessary to complete cache recovery and roll forward the database to the state it was in at the time of instance failure. Because changes to blocks in the undo segment are recorded in the redo log, rolling forward the redo log also regenerates the corresponding undo blocks that contain a record of changes that need to be undone when transaction recovery is run to roll back incomplete transactions.

Oracle can initiate the first pass of the recovery process concurrently with the GCS rebuild process. After the first pass completes, the database is made available for service to applications for all but the data blocks impacted by the failure [2] (that is, for all data blocks but those in the recovery set). The buddy instance mechanism can potentially eliminate the first pass thereby making the database available for service almost immediately.

4. BUDDY INSTANCE MECHANISM TO HANDLE SINGLE INSTANCE FAILURES

Under the buddy instance mechanism, each RAC instance becomes a protected instance by virtue of having a designated instance to serve as its buddy. In a two instance RAC database shown in Figure 4, instance-1 is designated to serve instance-2 as its buddy instance and instance-2 is designated to serve instance-1 as its buddy instance. This designation of buddy instances is referred to as *buddy instance map*. This is also called as *one-on-one buddy instance map* as each protected instance has one designated buddy instance.

As changes to the database are recorded at run-time by an instance in its redo log, a server process in its buddy instance continuously scans that redo to construct the recovery set. The server process starts at the checkpoint RBA (referred to as the s*tart RBA*) and scans the redo till the end of the log. By default, the rate at which this server process scans the redo is adjusted to the ongoing redo generation rate. This can be overridden by using the *_buddy_instance_num_read_buffers* parameter to establish a constant redo scan rate. This parameter dictates the number of buffers, each of size 4MB, which will be read and processed approximately every three seconds. If the value of the parameter is low, the server process scans less aggressively. This results in less load on the system but has the disadvantage that the amount of work required by the first pass of instance recovery may increase.

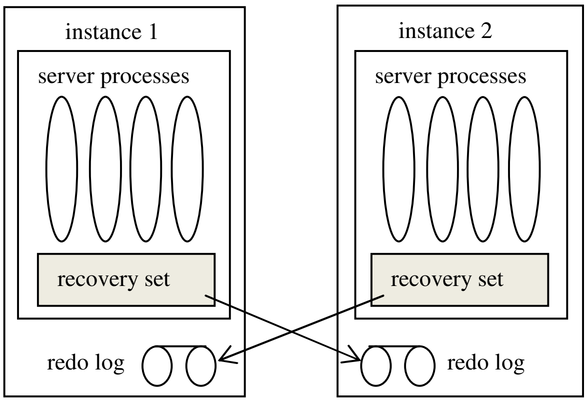


Figure 4. Buddy instance map in two instance RAC database

At regular intervals, the server process estimates the amount of time required for the first pass of instance recovery, if a crash were to happen at that time. If that amount of time is less than the value of the *_buddy_instance_scan_phase_threshold* parameter (which has a default value of 3 seconds), the redo is not scanned.

For each block in the recovery set, Oracle maintains the *last RBA* which refers to the RBA of the last redo record that changed or created that block. If defined, this RBA must be between the start RBA and the end of the log.

4.1 Recovery Set Pruning

As the checkpoint of an instance progresses, its buddy instance must advance its start RBA to that of the instance's checkpoint RBA. After advancing this, the recovery set can be pruned by removing blocks which have a last RBA that is less than the new start RBA.

4.2 RAC Membership Changes

Oracle RAC Database maintains the buddy instance map and automatically updates it as and when instances join or leave the cluster. Instances can be added or taken out of an Oracle RAC system without shutting the database down. When an instance joins or leaves the cluster, the buddy instance map must be dynamically adjusted to reflect the new cluster configuration.

When instance-3 joins the Oracle RAC system shown in Figure 4, the buddy instance map is updated to that shown in Figure 5. Here, instance-1 is designated to serve instance-2 as its buddy instance, instance-2 is designated to serve instance-3 as its buddy instance and instance-3 is designated to serve instance-1 as its buddy instance.

In the similar way, when instance-3 leaves the Oracle RAC system shown in Figure 5, the buddy instance map is updated to that shown in Figure 4.

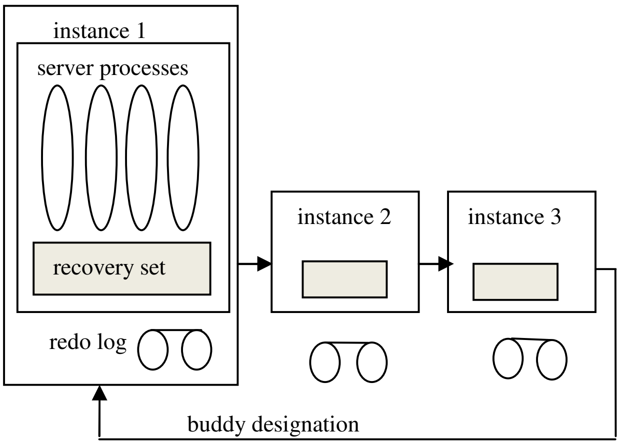


Figure 5. Buddy instance map in three instance RAC database to handle single instance failure

4.3 Handshake with Instance Recovery

When an instance fails, its buddy instance is asked to perform instance recovery of the failed instance. In the first pass of instance recovery, the buddy instance uses the recovery set that was constructed during run-time. If the checkpoint RBA of the recovery set that was constructed during runtime is behind the checkpoint RBA as determined by instance recovery, the buddy instance prunes the recovery set using the checkpoint RBA for instance recovery. If the buddy instance had not scanned till the end of the redo log prior to instance failure, it will do so during the first pass of instance recovery.

GCS can make the global cache available to surviving instances as soon as the recovery set is constructed by the first pass of instance recovery. Since Oracle is expected to spend significantly less in the first pass, the availability of the database is significantly increased by using the buddy instance mechanism.

5. EXTENDING BUDDY INSTANCE MECHANISM TO HANDLE MULTI-INSTANCE FAILURES

The buddy instance mechanism of Oracle RAC currently handles single instance failure only. This section presents a possible implementation of the buddy instance mechanism for multi-instance failures. This does not constitute a commitment by Oracle to deliver any code or functionality and should not be relied upon in making purchasing decisions.

One possible implementation to handle multi-instance failure is to give each protected RAC instance more than one buddy instance each continuously scanning the redo log of the protected instance during runtime to construct its recovery set. The number of instances that serve as buddies determines the degree of multi-instance failure that can be handled while fully getting the benefit of the buddy instance mechanism. In a RAC of n-instances, each instance can have up to (n-1) buddies.

The recommended buddies for an instance depend on statistics such as number of instances that previously failed together and which instances failed together. It is possible for Oracle Database to recommend the buddies for each of the RAC instances based on the statistics that were collected during previous instance failures.

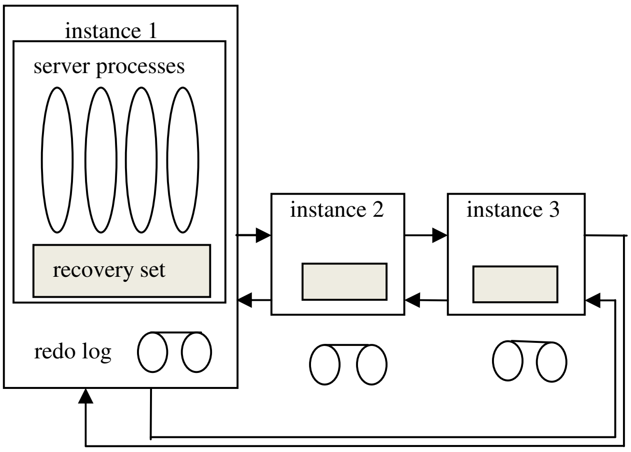


Figure 6. Buddy instance map in three instance RAC database to handle multi-instance failure

Figure 6 shows an example of a three instance Oracle RAC system in which a single instance is protected by two buddies. This system can tolerate up to two instances failing simultaneously, while still taking advantage of the buddy instance mechanism.

5.1 Recovery Set Pruning

For each protected RAC instance, all its buddy instances must perform the steps detailed in Section 4.1.

As the checkpoint of an instance progresses, all its buddy instances must advance the start RBA to that of the instance's checkpoint RBA and prune its recovery set by removing the blocks which have a last RBA that is less than the new start RBA.

5.2 RAC Membership Changes

This is an extension of the RAC membership changes described in Section 4.2, where instances can be added or taken out of an Oracle RAC system without shutting the database down.

When an instance joins or leaves the cluster configuration, the buddy map needs to be updated based on the statistics that were collected during previous instance failures with the same cluster configuration.

5.3 Handshake with Instance Recovery

When one or more instances fail, instance recovery is performed by a surviving buddy instance which has the recovery set for the most number of failed instances. This instance is designated the *recovery instance.* Below is the sequence of events that are performed on recovery instance during the first pass of recovery.

1. If the recovery instance is the buddy of a failed instance and it did not scan the redo till the end of the log prior to the failure, the recovery instance needs to do so now.
2. If the recovery instance does not have the recovery set for a specific failed instance, it needs to receive the recovery set from the buddy of that specific instance (if there is one). If that buddy instance had not scanned the redo till the end of the log prior to the failure, the recovery instance needs to do so now.
3. If a failed instance does not have a surviving buddy instance, the recovery instance needs to scan the entire redo of that failed instance from the checkpoint of the failed instance to the end of the log.
4. The recovery instance needs to merge the recovery sets of all the failed instances.

In the Oracle RAC system shown in Figure 6, if both instance-1 and instance-2 fail, instance recovery needs to be performed by instance-3 which has the recovery set for both of the failed instances.

6. PERFORMANCE STUDY

Experiments were conducted to measure the impact on database throughput and also to measure the acceleration in instance recovery.

6.1 Impact on Run Time Performance

Experiments were conducted using a TPC-C workload to evaluate the impact on database throughput with the buddy instance mechanism enabled.

6.1.1 *Hardware Setup*

The study was conducted using an Oracle Exadata Database Machine [6] with an InfiniBand cluster interconnection for Oracle RAC servers. An X3-2 RAC configuration was used, comprising two database server nodes, each equipped with 32, 8-core Intel Xeon processors running at 2.9 GHz and 128 GB of memory.

6.1.2 *TPC-C Workload*

TPC-C benchmark is the industry standard for evaluating the performance of OLTP systems [5]. The setup consisted of 1000 warehouses and 512 clients.

CPU utilization at 91%, was not affected when using the buddy instance mechanism. Not surprisingly however, the number of reads of the redo log increased.

Figure 7 shows the impact on database throughput for a fully-cached TPC-C workload (where the buffer cache is sized large enough to accommodate the entire database), and a partially-cached TPC-C workload (where the buffer cache is sized at approximately 20% of the database size). For a fully-cached TPC-C workload, no impact was observed on database throughput when the buddy instance mechanism was enabled. Due to the nature of the workload, around 1% variation is expected across different runs. The impact on database throughput was less than 2% for the partially cached TPC-C workload.

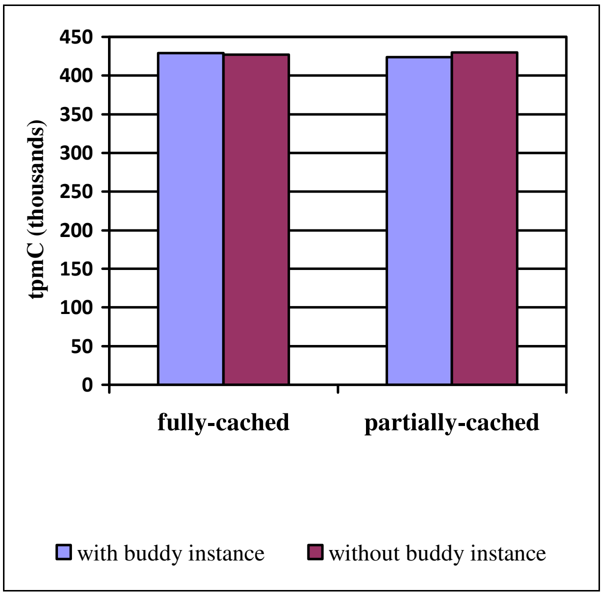


Figure 7. Impact on database throughput (tpmC) for a TPC-C workload

6.2 Acceleration in Instance Recovery

6.2.1 *Hardware Setup*

This study was also conducted using an Oracle Exadata Database Machine [6] with an InfiniBand cluster interconnection for Oracle RAC servers. An X2-8 RAC configuration was used, comprising two database server nodes, each equipped with 8, 12-core Intel Xeon processors running at 2.40 GHz, 2 TB of memory and 14 shared storage servers amounting to 200 TB total storage capacity over a Direct-to-Wire 3 x 36 port QDR (40 Gb/sec) InfiniBand interconnect.

6.2.2 *TPC-C Workload*

The TPC-C benchmark application was run to generate a workload. The FAST_START_MTTR_TARGET parameter [10] which affects the rate of checkpointing and hence the duration of instance recovery, was set to its default value of "0" in line with what is done on most customer systems. After running the TPC-C benchmark for 27 minutes, an instance was crashed. Figure 8 shows the time taken both by the first pass of instance recovery and the time taken overall by instance recovery. The time taken for instance recovery as a whole has been reduced because of the decrease in the time taken for the first pass.

Without making use of the buddy instance, instance recovery spent 130 seconds in its first pass when the second pass needed to apply 11.5 GB of redo. By comparison, when using the buddy instance mechanism, instance recovery spent only 1 second in its first pass when the second pass needed to apply a similar amount of redo. Since the first pass correlates with brown-out time, the time that the database was completely unavailable to applications was reduced from 130 seconds to 1 second.

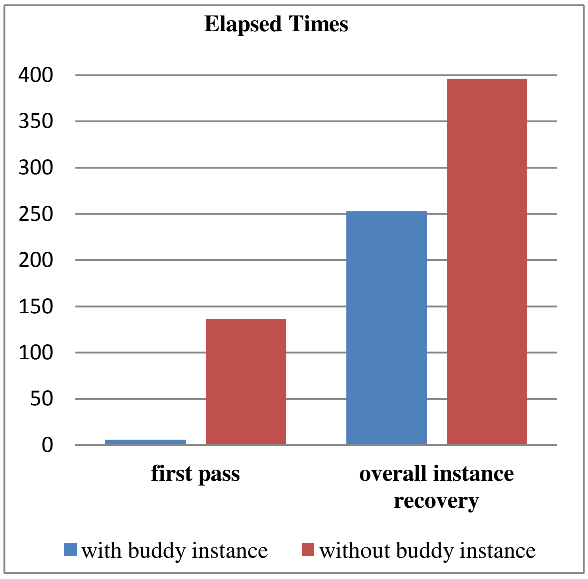


Figure 8. Elapsed times (seconds) for a TPC-C workload

6.2.3 *Results for a Commercial Workload*

The experiment was repeated using a real customer workload. The database consisted of 31 tables and 35 indexes. The total on-disk size of the database was approximately 200GB. This workload generated redo by having multiple concurrent users repeatedly perform updates and inserts into their own tables. An instance was then crashed. Figure 9 shows the time taken by the first pass and the time taken overall for instance recovery. This experiment had set the FAST_START_MTTR_TARGET [10] parameter to 100.

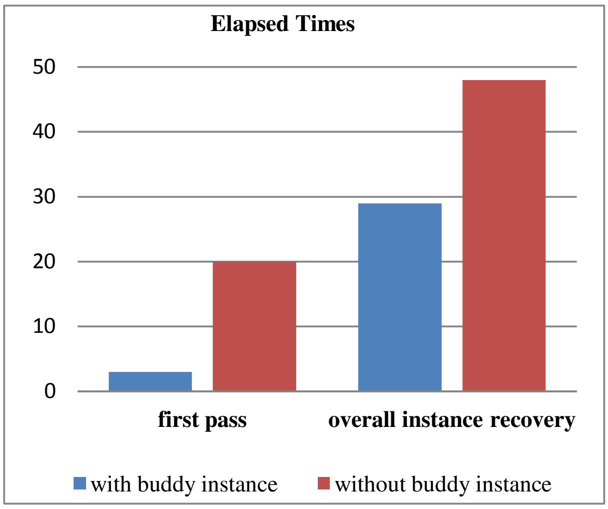


Figure 9. Elapsed times (seconds) for a commercial workload

Without using the buddy instance, instance recovery spent 20 seconds in its first pass when the second pass needed to apply 1.5GB of redo. By comparison, when using the buddy instance mechanism, instance recovery spent only 3 seconds in the first pass when the second pass needed to apply a similar amount of redo.

Oracle database became available on surviving instances in 3 seconds versus 20 seconds. In addition, the overall time for instance recovery was reduced from 48 seconds to 29 seconds. This experiment validates the key claims in this paper by reducing both the brown-out time and overall time for instance recovery.

7. RELATED WORK

The fast-start fault recovery [10] technology in Oracle Database allows control over the duration of the roll-forward phase by adaptively varying the rate of checkpointing. This is applicable more to crash recovery which takes place when every RAC instance fails. In Oracle RAC Database, each instance may have differing amounts of workload and different instances may perform checkpoint activity at different rates. The *database checkpoint* in RAC is the instance checkpoint that has the lowest checkpoint SCN of all the instances. For instance recovery, the instance checkpoint determines the set of redo log changes that need to be applied. Unlike fast-start fault recovery which relies on the database checkpoint, the buddy instance mechanism relies on the instance checkpoint and is therefore a more customized solution for instance recovery in Oracle RAC Database. Since checkpoint issues disk writes for data blocks, aggressive checkpoint activity can be detrimental for database throughput. The buddy instance mechanism only issues disk reads for the redo log which do not mandate acquisition of locks as no synchronization is required. The buddy instance mechanism is therefore less intrusive than using fast-start fault recovery. This paper recommends that the buddy instance mechanism be used in conjunction with fast-start fault recovery technology for best results.

Regarding other database vendors that have a shared disk cluster database solution, published documentation for both SAP Sybase ASE Cluster Edition [11] and IBM DB2 pureScale Clustered Database [12] indicates that neither make use of a scheme similar to the buddy instance mechanism.

8. CONCLUSION

The availability of a cluster system can be improved by making use of the buddy instance mechanism[1] which significantly reduces the amount of time the database spends in the first pass of instance recovery. Since the database can be made available as soon as the first pass of instance recovery completes, the availability of the cluster increases significantly. In addition, the overall time taken for instance recovery is reduced.

9. REFERENCES

[1] Oracle Corporation. Oracle 9i Real Application Clusters concepts Release 2 (9.2), Part Number A96597-01.

[2] Building Highly Available Database Servers Using Oracle Real Application Clusters, An Oracle White Paper, May 2002.

[3] Lahari, T., Srihari, V., Chan, W., Macnaughton, N., Chandrasekaran, S. Cache Fusion: Extending Shared-Disk Clusters with Shared Caches. *Proceedings of the VLDB Conference 2001.*

[4] Joshi, A., Bridge, Loaiza, J., W., Lahiri, T. Checkpointing in Oracle. *Proceedings of the 24th VLDB conference 1998.*

[5] The Transaction Processing Council. TPC-C Benchmark. http://www.tpc.org/tpcc/, 2016.

[6] Oracle Exadata Database Machine System Overview 12*c* Release 1 (12.1), Part Number E51953-14, Feb 2017

[7] Oracle Database Concepts 12c Release (12.1), Part Number E41396-13.

[8] Bridge, W., Joshi, A., Keihl, M., Lahiri, T., Loaiza, J. The Oracle Universal Server Buffer Manager. Proceedings of the 23rd VLDB Conference, pp. 590-594, 1997.

[9] Lahari, T., Ganesh, A., Weiss, R., Joshi, A. Fast-Start: quick fault recovery in oracle, Proceedings of the ACM SIGMOD international conference on Management of data, 2001.

[10] Oracle9*i* Database Performance Tuning Guide and Reference Release 2 (9.2), Part Number A96533-02.

[11] Technical Comparison of Oracle Real Application Clusters 11g vs. IBM DB2 v9 for Linux, Unix, and Windows, An Oracle White Paper, December 2009.

[12] Adaptive Server® Enterprise Cluster Edition 15.5 Users Guide, Document ID: DC00768-01-1550-01

[1] The technique presented in the paper has been granted a U.S. patent.

DBaaS Cloud Capacity Planning - Accounting for Dynamic RDBMS Systems that Employ Clustering and Standby Architectures

Antony S. Higginson
Oracle Advanced Customer Support Services
Didsbury, Manchester, UK
antony.higginson@oracle.com

Norman W. Paton
School of Computer Science
University Of Manchester
Manchester, M13 9PL, UK.
norman.paton@manchester.ac.uk

Suzanne M. Embury
School of Computer Science
University Of Manchester
Manchester, M13 9PL, UK.
suzanne.m.embury@manchester.ac.uk

Clive Bostock
Oracle Advanced Customer Support Services
Didsbury, Manchester, UK
clive.bostock@oracle.com

ABSTRACT

There are several major attractions that Cloud Computing promises when dealing with computing environments, such as the ease with which databases can be provisioned, maintained and accounted for seamlessly. However, this efficiency panacea that company executives look for when managing their estates often brings further challenges. Databases are an integral part of any organisation and can be a source of bottlenecks when it comes to provisioning, managing and maintenance. Cloud computing certainly can address some of these concerns when *Database-as-a-Service* (DBaaS) is employed. However, one major aspect prior to adopting DBaaS is Capacity Planning, with the aim of avoiding *under-estimation* or *over-estimation* of the new resources required from the cloud architecture, with the aim of consolidating databases together or provisioning new databases into the new architecture that DBaaS clouds will provide. Capacity Planning has not evolved sufficiently to accommodate complex database systems that employ advanced features such as Clustered or Standby Databases that are required to satisfy enterprise SLAs. Being able to efficiently capacity plan an estate of databases accurately will allow executives to expedite cloud adoption quickly, allowing the enterprise to enjoy the benefits that cloud adoption brings. This paper investigates the extent to which the physical properties resulting from a workload, in terms of CPU, IO and memory, are preserved when the workload is run on different platforms. Experiments are reported that represent OLTP, OLAP and Data Mart workloads running on a range of architectures, specifically single instance, single instance with a standby, and clustered databases.

This paper proposes and empirically evaluates an approach to capacity planing for complex database deployments.

Keywords

Cloud, DBaaS, Capacity Planning, Database, Provisioning, Standby, Clustering

1. INTRODUCTION

Traditionally, companies accounted for the cost of assets associated with their I.T. using Capex (Capital Expenditure) type models, where assets such as hardware, licenses and support, etc, were accounted for yearly. For example, a software license usually is based on an *on-premises* model that would be user based, or by the CPU if the application served many thousands of users. The advent of Cloud computing, with the *pay-as-you-go* subscription based model, has changed the way company executives look at the costing models of their I.T.

A similar paradigm unfolds when I.T. departments such as Development, Delivery and Support teams need to provision environments quickly to meet their business goals. Traditional project methodologies would request environments aiding development and testing with the goal of going live. Procurement and provisioning took time that was often added to the project lifecycle. Cloud computing addresses such issues so that a user can, with ease, request the rapid provision of resources for a period of time.

Once the system went live those Delivery and Support teams would then need to account for resources those particular systems consumed, reconciling with the Line Of Business (LOB). The results of that analysis would then feed back into next year's Capex model. This ongoing capacity planning to assess if they have enough resources is needed to ensure is that, as systems grow, there are enough resources to ensure that the system is able to meet QoS (Quality of Service) expectations. Cloud Computing has also made some advances here by enabling a metering or charge-back facility that can accurately account for the resources used (CPU, Memory, Storage). Cloud Computing can dynamically modify the cloud to reduce or increase those resources

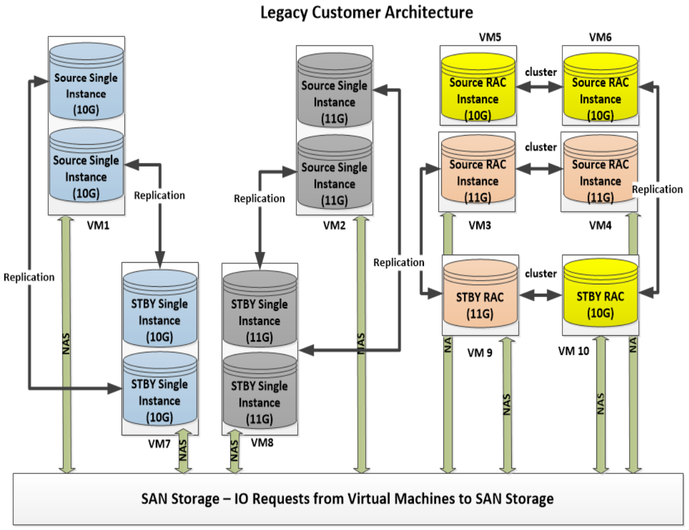

Legacy Customer Architecture
Source Single Instance (10G)
Source Single Instance (10G)
VM1
Replication
Replication
Source Single Instance (11G)
Source Single Instance (11G)
VM2
Replication
VM5
VM6
Source RAC Instance (10G)
cluster
Source RAC Instance (10G)
Source RAC Instance (11G)
cluster
Source RAC Instance (11G)
Replication
VM3
VM4
Replication
STBY RAC (11G)
cluster
STBY RAC (10G)
VM 9
VM 10
STBY Single Instance (10G)
STBY Single Instance (10G)
VM7
STBY Single Instance (11G)
STBY Single Instance (11G)
VM8
NAS
SAN Storage – IO Requests from Virtual Machines to SAN Storage

Figure 1: Example Architecture: Typical customer legacy database architecture.

as needed by the client.

However, companies with large estates have the additional challenge of having a plethora of database versions, for example, each database version offering a different feature that has a performance benefit over another database version. Similarly, the databases may be running on a eclectic set of operating systems and hardware, each affecting the workload in a subtle or major way. For example, the latest running version of a database may run on a highly configured SAN utilising the latest techniques in query optimization and storage. Comparing this footprint with an older version of software and infrastructure often leads to a *Finger-in-the-air* type approach.

A key feature of DBaaS is the ability to multi-tenant those databases where different workloads and database configurations can coexist in the shared resources, adding to the challenge of making effective capacity planning decisions. Determining the allocation is further complicated if the database utilises advanced features such as Clustering or Failover Technology, as workloads shift from one instance to another or are shared across multiple instances based on their own resource allocation managers. Furthermore, if a database employs a standby, this further complicates capacity planning decisions.

Cloud Computing is in its infancy, with incremental adoption within the industry as companies try and determine how to unpick their database estates and move them to cloud infrastructure. Databases often grow organically over many years in terms of their data and complexity, which often leads to major projects being derived when a major upgrade or re-platform exercise is required. With the introduction of cloud these exercises are becoming more prudent. This often leads to a series of questions on Capacity Planning.

- What is the current footprint of the database including any advanced features such as Standby or Clustering?
- What is the current configuration of the database?
- What type of DBaaS should I create?
- What size of DBaaS should I create?
- Can I consolidate databases that have similar configuration and utilisation foot-prints?
- Will my SLAs be compromised if I move to a cloud?

Such questions become very important prior to any provisioning or migration exercise.

The time taken to perform this analysis on databases also has a major impact on a company's ability to adopt cloud technologies often squeezing the bandwidth of the delivery and support teams. The departments suffer *paralysis-by-analysis*, and the migration to the cloud becomes more protracted to the frustration of all involved. If the analysis is not performed accurately then the risks of *over-estimation* and *under-estimation* increase. Being able to automate the gathering of data, analysing the data and then making a decision becomes ever more important in enterprises with large estates.

In this paper we look at the challenges of Capacity Planning for advanced database systems that employ clustering and standby databases, with a view to migration to a cloud. Our hypothesis is: "That a model based on physical measures can be used to provide dependable predictions of performance for diverse applications". We make two main contributions:

1. We propose an approach to workload analysis based on physical metrics that are important to capacity planning for database systems with advanced configurations.
2. We report the results of an empirical analysis of the metrics for several representative workloads on diverse real-life configurations.

The remainder of the paper is organised as follows. Section 2 introduces the Background and Related Work. In Section 3 we detail the environmental setup for conducting experiments outlining the database capacity planning problem. In Section 4 we introduce our solution in detail and provide details on the experiments and analysis. Section 5 gives conclusions and future work.

2. BACKGROUND AND RELATED WORK

2.1 Background

Fig 1 shows an example environment of a company that is running different versions and configurations of databases on VM hardware. Physical machines are dissected into 10 VM's giving a level of separation. On these 10 VM's a total of 12 databases are run, of which 6 are primary databases and 6 are standby databases. This MAA *(Maximum Availability Architecture)* allows the company some comfort by running their primary *(Platinum)* SLA level applications on VM numbers 3, 4, 5 and 6, which host two clustered databases (offering a degree of resilience against node failure). In addition, these clustered databases have a physical standby database running on VM's 9 and 10 in case of database failure or corruption. Similarly, the 4 single instance stand alone databases that are running on VM's 1 and 2 also have a replicated standby database running on VM's 7 and 8, again offering the company some comfort that their secondary *(Gold)* level of applications will have a standby database for failover, should they need it.

The company also wish to increase their ROI *(Return on Investment)* with this environment and thus often open up the standby databases in *"Read Only"* mode during special times for applications that need to run year-end or month-end type BI *(Business Intelligent)* reports. This particular type of architectural pattern is a typical configuration companies use today to manage their database environments and applications that have 24*7 type SLAs. The difficulty becomes apparent when a new exercise is introduced that looks at consolidating, upgrading and migrating those environments listed in Fig 1 to a new cloud architecture, where resources can be tightly accounted and dynamically assigned. We are then faced with a capacity planning exercise.

2.2 Related Work

The objective of capacity planning is to provide an accurate estimate of the resources required to run a set of applications in a database cloud. Achieving this answer relies on the accurate capture of some base metrics, based on historical patterns, and applying some modelling techniques to form a prediction. There are two main viewpoints: the viewpoint of the CSP *(Cloud Service Provider)* in what they offer and their capabilities, i.e are there enough resources to provide services to consumers; and the viewpoint of the consumer, for example, can a customer capacity plan their systems against the CSP's capability? Indeed if the customer wishes to become a CSP but in a private cloud configuration, then the first viewpoint also becomes important.

A CSP offers resources, and existing models use various techniques to help customers assess the CSP capabilities. MCDM (Multi Criteria Decision Making) weighs the attributes of an individual database by their importance in helping to choose the right cloud (Mozafari et al 2013 [16] and Shari et al 2014 [19]. CSP's can also be assessed using a pricing model to validate their capability based on a consumers single systems workload as suggested by (Shang et al [20]); using this financial approach contributes to the *value-for-money* question that many enterprises seek when deciding on the right cloud.

If a consumer has a cloud, knowing where to place the workload based on utilisation to achieve the best fit is critical when beginning to answer the QoS (Quality of Service) question, and techniques such as *bin-packing* algorithms (Yu et al [21]) help achieve this answer. However systems may have dynamic workloads, which may evolve organically as datasets and/or numbers of users grow or shrink, as is especially common in internet based systems. There is a need for constant assessment of said workloads. Hacigumns et al [10] and Kouki et al [11] both look at the workload of an application or the query being executed, and then decide what type of database in a cloud would satisfy QoS. Mozafari et al [15] suggests using techniques that capture log and performance data over a period of time, storing them in a central repository, and modelling the workloads at a database instance level. With the advent of Virtualisation that enterprises utilise, including CSP's, when running their estates, several techniques such as coefficient of variation and distribution profiling are used to look at the utilisation of a Virtual Machine to try and capacity plan. Mahambre and Chafle [13] look at the workload of a Virtual Machine to create relationship patterns of workloads to understand how resources are being utilised, analysing the actual query being executed to predict if and when it is likely to exhaust resources available.

There seems to be a consensus among several academics (Shang et al [20], Loboz [12] and Guidolin et al 2008 [9]) on the need for long term capacity planning and the inadequacy of capacity planning in this new age of cloud computing using current techniques. The techniques used today assume that the architecture is simple, in that the architecture does not utilise virtualisation or advanced database features such as standby's and clustering technology, but in the age of consolidation and drive for standardisation, the architecture is not simple. Enterprises use combinations of technology in different configurations to achieve their goals of consolidation or standardisation. Most models use a form of linear regression to predict growth patterns. Guidolin et al 2008 [9] conducted a study of those linear regression models and came to the conclusion that as more parameters are added the models become less accurate, something also highlighted by Mozafari et al 2013 [15]. To mitigate against this inaccuracy more controls are added at the cost of performance of the model itself. For example, predicting the growth of several databases based on resource utilisation may become more inaccurate as the number of source systems being analysed increases, therefore requiring more controls to keep the accuracy. This is certainly interesting when trying to capacity plan several applications running on different configurations prior to a migration to a cloud. In addition, trying to simulate cloud computing workloads to develop new techniques is also an issue; Moussa and Badir 2013 [14] explained that the TPC-H [4] and TPC-DS [3] benchmarks are not designed for Data Warehouses in the cloud, further adding to the problem of developing and evaluating models.

3. EXPERIMENTAL SETUP

Given a description of an existing deployment, including the Operating System, Database and Applications running on that database (Activity), a collection of monitors on the existing deployment that report on CPU, Memory, IOPS's and Storage, the goal is to develop models of the existing configuration that contain enough information to allow reliable estimates to be made of the performance of a deploy-

Workload Type	Workload Profile	DBNAME(S)	Workload Description	Number of Users	Duration (hh:mi)	Avg Transaction per sec
OLTP	General usage	RAPIDKIT RAPIDKIT2 DBM01	General Online Application with updates, inserts and deletes simulate working day	100 2000 (DBM01)	23:59	0.2
OLTP	Morning Peak Logon Surge	RAPIDKIT RAPIDKIT2 DBM01	Morning Surge to simulate users logging on to the Online Application with updates, inserts and deletes	100 1000 (DBM01)	2:00	0.2
OLTP	Lunch Time Peak Logon Surge	RAPIDKIT RAPIDKIT2 DBM01	Lunch Time Surge to simulate users logging on to the Online Application with updates, inserts and deletes	100 1000 (DBM01)	1:00	0.2
OLTP	Evening Time Peak Logon Surge	RAPIDKIT RAPIDKIT2 DBM01	Evening Time Surge to simulate users logging on to the Online Application with updates, inserts and deletes	100 1000 (DBM01)	5:00	0.2
Daily OLTP Hot Backup taken at 23:00						
OLAP	Data Warehouse General Usage	RAPIDKIT RAPIDKIT2 DBM01	General Data Warehousing Application with heavy Selects taking place out of hours building Business Intelligence data	5 400 (DBM01)	8:00	0.4
Daily OLAP Hot Backup taken at 06:00						
Daily OLAP archivelog backups taken at 12:00,18:00,00:00						
DM	OLTP General Usage	RAPIDKIT RAPIDKIT2 DBM01	Combination of DML taking place during the business day and heavy DML taking out of ours	200 1000 (DBM01)	23:59	0.2
DM	OLTP Morning Peak Logon Surge	RAPIDKIT RAPIDKIT2 DBM01	Morning Surge to simulate users logging on to the Online Application with updates, inserts and deletes	100 500 (DBM01)	2:00	0.2
DM	OLTP Lunch Time Peak Logon Surge	RAPIDKIT RAPIDKIT2 DBM01	Morning Surge to simulate users logging on to the Online Application with updates, inserts and deletes	100 500 (DBM01)	2:00	0.3
DM	OLTP Evening Time Peak Logon Surge	RAPIDKIT RAPIDKIT2 DBM01	Evening Time Surge to simulate users logging on to the Online Application with updates, inserts and deletes	100 500 (DBM01)	5:00	0.3
DM	OLAP Batch Loads Peak	RAPIDKIT RAPIDKIT2 DBM01	Evening Time Surge to simulate users logging on to the Online Application with updates, inserts and deletes	5 400 (DBM01)	8:00	0.3
Daily DM Hot Backup taken at 06:00						
Daily DM archivelog backups taken at 12:00,18:00,00:00						

Table 1: Database Workloads

ment when it is migrated to a cloud platform that may involve a Single Database, a Clustered Database or Standby Databases. To meet this objective, we must find out if a workload executed on one database is comparable to the same workload running on the same database on a different host.

Our approach to this question is by way of empirical evaluation. Using increasingly complex deployments, of the type illustrated in Fig 2, and representative workloads, we establish the extent to which we can predict the load on a target deployment based on readings on a source deployment. This section describes the workloads and the platforms used in the experiments.

3.1 Workloads

A Workload can be described as the activity being performed on the database at a point-in-time, and essentially is broken down into the following areas:

- *Database* - An Oracle database is a set of physical files on disk(s) that store data. Data may be in the form of logical objects, such as tables, Views and indexes, which are attached to those tables to aid speed of access, reducing the resources consumed in accessing the data.

- *Instance* - An Oracle instance is a set memory structures and processes that manage the database files. The instance exists in memory and a database exists on disk, an instance can exist without a database and a database can exist without an instance.

- *Activity* - The DML (Data Modification Language)/DDL (Data Definition Language) i.e. SQL that is being executed on the database by the application, creates load consisting of CPU, memory and IOPS/s.

.

The monitors used to capture the data report on IOPS's (Physical reads and Physical Writes), Memory (RAM assigned to a database or host) and CPU (SPECINT's). SPECInt is a benchmark based on the CINT92, which measures the integer speed performance of the CPU, (Dixit) [6]. The experiments involve controlled execution of several types of workloads on several configurations of database. Moussa and Badir 2013 [14] describe how running of controlled workloads using TPC has not evolved for clouds, therefore we will use a utility called *swingbench* (Giles)[8] to generate a controlled load based on TPC-C [5]. The workload is generated on several Gb's of sample data based on the Orders Entry (OE) schema that comes with Oracle 12C. The OE schema is useful for dealing with intermediate complexity and is based on a company that sells several products such as software, hardware, clothing and tools. Scripts are then executed to generate a load against the OE schema to simulate DML transactions performed on the database of a number of users over a period of Hour.

3.2 Outline of the Platforms

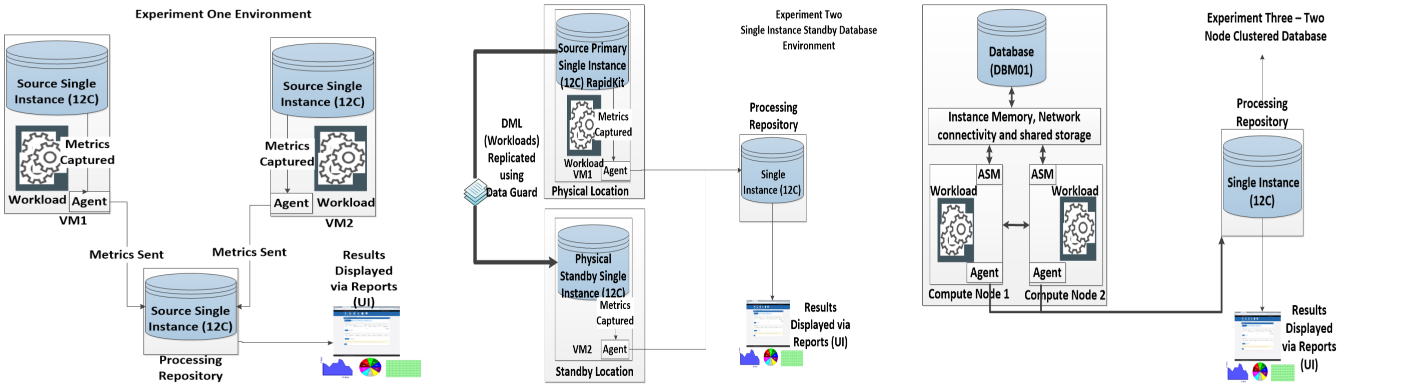
Three different types of workload were created *(OLTP, OLAP and Data Mart)* as shown in Table 1. The Database is placed in archivelog mode during each execution of the workload further creating IO on the Host and allowing for a hot backup to be performed on the database. The backup acts as a *'houskeeping'* routine by clearing down the archivelogs to ensure the host does not run out of storage space. This type of backup routine is normal when dealing with databases and each backup routine is executed periodically depending upon the workload.

4. EXPERIMENTS AND ANALYSIS

A number of experiments were conducted to investigate if a workload executed on one machine consumes similar resources when the workload is executed on another environment. The aim was to investigate what could cause dif-

VM Name	OS Type	CPU Details	Memory	Storage	Database Type	Products and Versions
Single Database Instance Configuration						
Virtual Machine 1	OEL Linux 2.6.39	4 * 2.9 Ghz	32Gb	300Gb	Oracle Single Instance Database (RapidKit)	• Enterprise Edition (12.1.0.2), • Data Guard (12.1.0.2), • Enterprise Manager Agent (12.1.0.4),
Virtual Machine 2	OEL Linux 2.6.39	4 * 2.9 Ghz	32Gb	300Gb	Oracle Single Instance Database (RapidKit2)	• Enterprise Edition (12.1.0.2), • Data Guard (12.1.0.2), • Enterprise Manager Agent (12.1.0.4),
Clustered Database Instance Configuration						
Clustered Compute Node 1	OEL Linux 2.6.39	24 * 2.9 Ghz	96Gb	14Tb	Oracle Clustered Multi-tenant Database Instance (DBM011)	• Enterprise Edition (12.1.0.2), • Data Guard (12.1.0.2), • Enterprise Manager Agent (12.1.0.4, • Grid Infrastructure (12.1.0.2), • Oracle Automatic Storage Manager (12.1.0.2),
Clustered Compute Node 2	OEL Linux 2.6.39	24 * 2.9 Ghz	96Gb	14Tb	Oracle Clustered Multi-tenant Database Instance (DBM012)	• Enterprise Edition (12.1.0.2), • Data Guard (12.1.0.2), • Enterprise Manager Agent (12.1.0.4, • Grid Infrastructure (12.1.0.2), • Oracle Automatic Storage Manager (12.1.0.2),
Standby Database Instance Configuration						
Virtual Machine 3	OEL Linux 2.6.39	4 * 2.9 Ghz	32Gb	1Tb	Oracle Single Instance Standby Database (STBYRapidKit, STBYRapidKit2)	• Enterprise Edition (12.1.0.2), • Data Guard (12.1.0.2), • Enterprise Manager Agent (12.1.0.4),
Central Repository Details						
Storage Repository	OEL Linux 2.6.39	24 * 2.4 Ghz	32Gb	500Gb	Oracle Single Instance Database (EMREPCTA)	• Enterprise Edition (11.2.0.3), • Enterprise Manager R4 including Webserver and BIPublisher (12.1.0.4), • Enterprise Manager Agent (12.1.0.4),

Table 2: Platform Outline


Experiment One Environment
Source Single Instance (12C)
Metrics Captured
Workload
Agent
VM1
Source Single Instance (12C)
Metrics Captured
Agent
Workload
VM2
Metrics Sent
Metrics Sent
Source Single Instance (12C)
Processing Repository
Results Displayed via Reports (UI)
Experiment Two Single Instance Standby Database Environment
Source Primary Single Instance (12C) RapidKit
Metrics Captured
Workload VM1
Agent
Physical Location
DML (Workloads) Replicated using Data Guard
Physical Standby Single Instance (12C)
Metrics Captured
VM2
Agent
Standby Location
Processing Repository
Single Instance (12C)
Results Displayed via Reports (UI)
Database (DBM01)
Instance Memory, Network connectivity and shared storage
ASM
ASM
Workload
Workload
Agent
Agent
Compute Node 1
Compute Node 2
Experiment Three – Two Node Clustered Database
Processing Repository
Single Instance (12C)
Results Displayed via Reports (UI)

(a) Single Instance (b) Single Instances with Standby Databases (c) Two Node Clustered Database

Figure 2: Experiment Architecture: different database combinations used for experiments.

ferences in the consumption of resources between workloads. The experiment focused on three types of database configuration:

- *Experiment 1* - Running three workloads *(OLTP,OLAP and DM)* on a single instance database.
- *Experiment 2* - Running the workloads *(OLTP,OLAP and DM)* on a single instance database with a Physical Standby Database.
- *Experiment 3* - Running the three workloads *(OLTP,OLAP and DM)* on a two node clustered database.

The database was always the same version between each host, the data set was always the same size to start, the workload was always repeatable in that the workload could be executed, stopped, the database reset and the same workload replayed.

4.1 Experimental Methodology

The experiments involve an eclectic set of hardware configured to run several different types of database as shown in Table 2. An agent polls the database instance every few minutes for specific metrics namely; Database Instance Memory, IOPS's (physical Reads/Writes) and CPU per sec. The metric results are stored in a central repository database, and are aggregated at hourly intervals. The configuration of the hardware, such as CPU Make model and SPECInt, and the database configuration are also stored in a central repository, which is then used as lookup data when performing comparisons between the performance of one workload on one database with the same workload on another database.

4.2 Experiment One - Single Database Instance

The first experiment was to execute three workloads on one single instance database on a virtual host (VM1) and then execute the same three workloads on another single instance database on another virtual host (VM2) as shown in Fig 2a. The database configurations were the same in Instance Parameters, Software Version and Patch Level. The Hardware configurations were the same in OS Level, Kernel Version, and memory configuration. Some differences exist in the underlying architecture such as the Physical hardware and the Storage as these where VM's created on different physical machines. We capture the metrics for each workload and analyse the extent to which physical properties are consistent across platforms. This is shown graphically in Fig 3

4.3 Results and Analysis Experiment One - OLTP Workload

The results for OLTP, covering Memory, CPU and IOPS/s are shown graphically in Fig 3. These are simple line graphs from the OLTP workload shown in Table 1. It was observed that the OLTP workload from a CPU perspective had several distinguishing features. It clearly shows that the workload starts off low until the beginning of the experiment where a sudden jump takes place and the OLTP workload begins. Then there is a general plateau that relates to the 24 hour periods and at various times from there on in there are spikes.

- *CPU utilisation* - CPU over a 72 hour period was not the same between the two databases but at it largest peak (evening surge) there was a difference of approximately 300 SPECInts or +88% (day 24 hour 11) in its utilisation. The difference in utilization between the two workloads without the peaks was approximately +20%.
- *CPU Spikes (Backup)* - There were several spikes in CPU at 00:00 - 02:00 and relate to the daily hot RMAN backup that is taken for the databases.
- *CPU Spikes (Morning Surge)* - A large CPU spike was observed for several hundred users accessing the database at 08:00.
- *IOPS/s (general)* - There is a large difference in IOPS (day 23 hour 9) where the difference at peak is +88%. The difference in general usage (i.e. without the peaks) was +7%.

CPU, Memory and IOPS/s over a 72 hour period show similar traits in that the workload begins and there is a jump in the activity as the users logon. The first set of results show that even when executed on similar platforms, the metrics for the OLTP workloads can be substantially different, especially in the CPU and IOPS utilisation.

4.4 Results and Analysis Experiment One - OLAP Workload

The results for OLAP covering Memory, CPU and IOPS/s are shown graphically in Fig 4. The difference between the OLTP and OLAP workload is that the OLAP workload is high in Select statements and the result set is larger. The IO is representative of a Data Warehouse building cubes for interrogation by a Business Intelligence reporting tool. The execution times for the workload are also different; OLTP is fairly constant in its usage, whereas OLAP is more concentrated out of normal working hours. It was observed that the OLAP workload runs out of hours for a period of around five hours and this matches the description shown in Table 1.

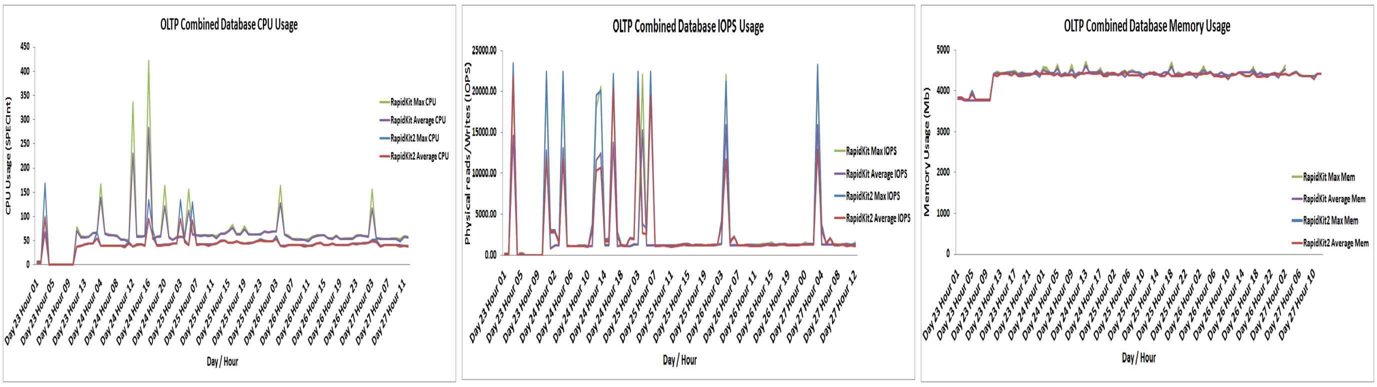
- *CPU Spikes (General Usage)* - CPU over a 72 hour period was not the same for the two databases, but at it largest peak there was a difference of only +1% (day 17 hour 05) in utilisation. Two workloads outside the peaks were essentially the same.
- *IOPS/s utilisation* - IOPS over a 72 hour period had a difference of approximately +50% in utilisation (Day 16 Hour 8); outside the peaks (Day 16 Hour 19) the utilisation is 0%.
- *IOPS/s Spikes (Backup)* - There are four backups that run during the 24 hours. Three of those backups are used as housekeeping routines that backup and delete the archivelogs; these backups are executed at 12:00, 18:00 and 00:00. One backup backs up the database (level-0) and the associated archivelogs, and this is executed at 06:00. There was no spike for 18:00 because the backup at 12:00 had removed the archivelogs and thus there was nothing to backup.

The OLAP Memory chart also showed the same characteristics as the IOPS/s and CPU charts in that there is a uniform pattern to there being a plateau and a spike over the 72 hours. Each of the databases had a memory configuration of 3.5Gb, given the OLAP workload would have had SQL requiring larger memory than 3.5Gb for sorting, thus sorts would have gone to disk rather than memory, accounting for the higher IOPS's readings in Fig 4 than in Fig 3.

4.5 Results and Analysis Experiment One - DataMart Workload

The results for the Data Mart covering Memory, CPU and IOPS/s are shown in Fig 5. It was observed that the Data Mart workload from a CPU perspective had several distinguishing features. It clearly shows that the workload starts off as the users connect and the workload is running, a sudden jump takes place at Day 10 Hour 3 as the Batch Loads are executed for approximately 6 hours, and this is repeated twice more throughout the 72 hours. There are also other peaks and troughs observed and these are consistent with the workload described in Table 1.

- *CPU utilisation* - CPU over a 72 hour period between the two databases and had a difference of approximately +64% during the normal day (Day 9 Hour 21). When the batch loads ran (Day 11 Hour 05) the difference in utilisation was +1%.
- *CPU Spikes (General)* - generally, the CPU utilisation between the two databases was the same, there is a difference of +1% at peak times.
- *IOPS/s Utilisation* - IOPS at peak (Day 9 Hour 21) had a difference of approximately +24%
- *Memory utilisation* - Memory was the same in general footprint however there were differences at peaks times of 300mb or +4%

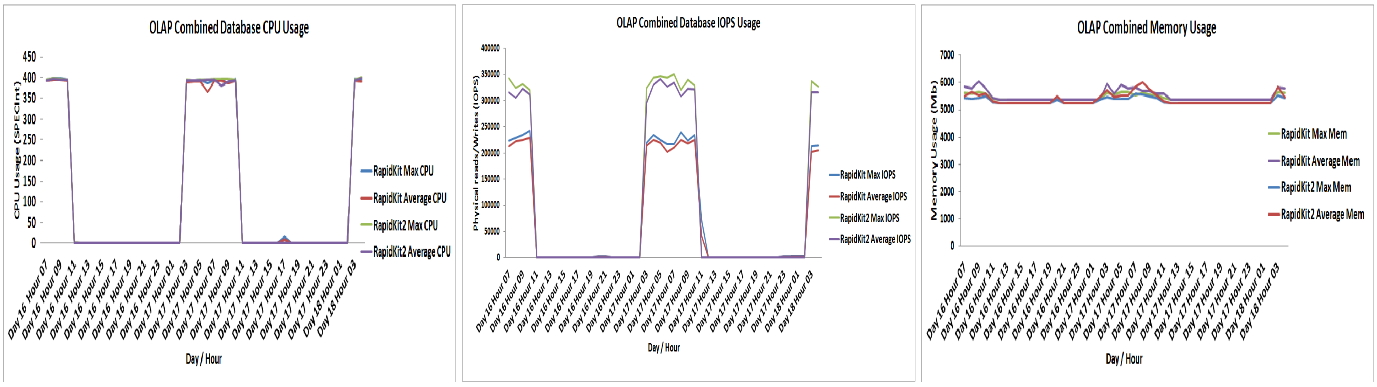


(a) CPU 72 hours

(b) IOPS's 72 Hours

(c) Memory 72 Hours

Figure 3: Results Single Instance OLTP: workload patterns for the 72 hour period.

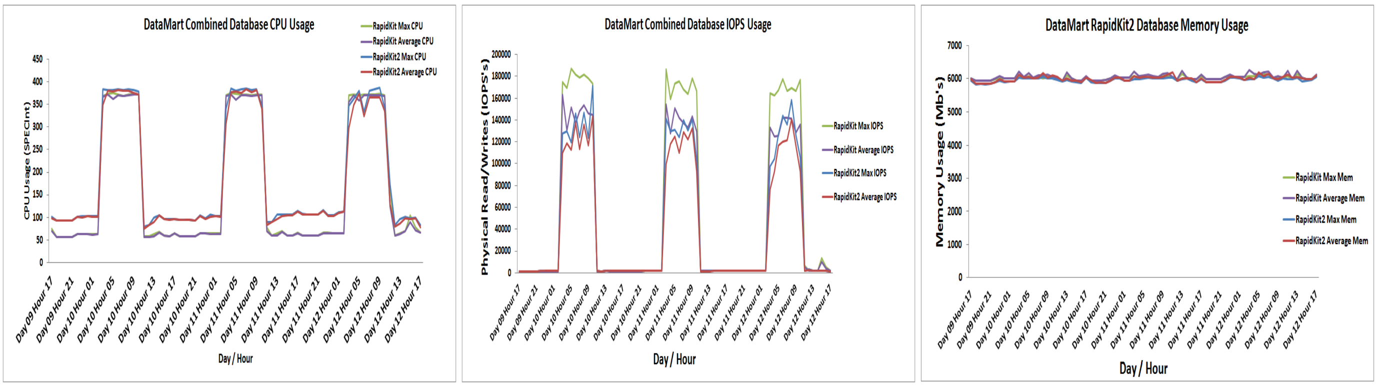


(a) CPU 72 hours

(b) IOPS's 72 Hours

(c) Memory 72 Hours

Figure 4: Results Single Instance OLAP: workload patterns for the 72 hour period.



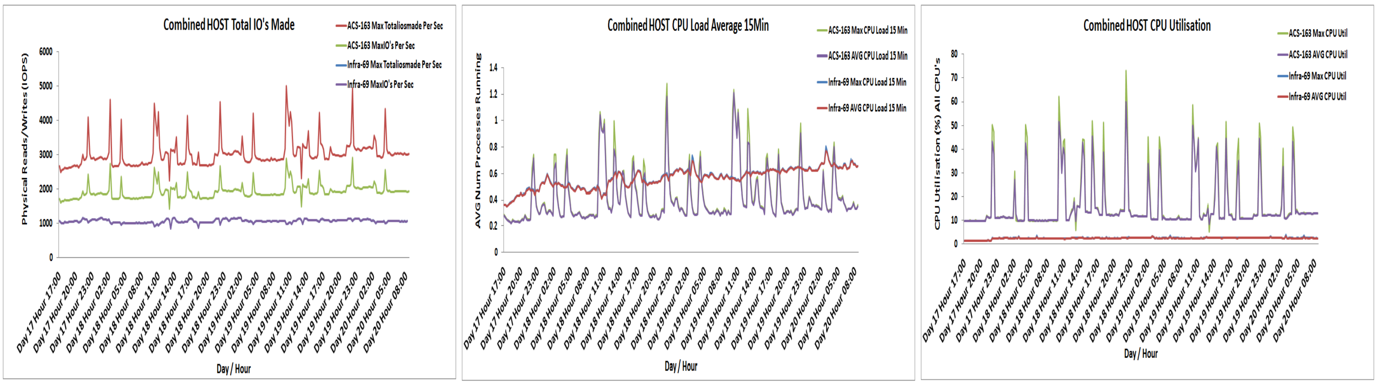
(a) CPU 72 hours

(b) IOPS's 72 Hours

(c) Memory 72 Hours

Figure 5: Results Single Instance Data Mart: workload patterns for the 72 hour period.

In general there is a difference in the VM's at a CPU level. The VM named *acs-163* has a configuration of 16 Threads(s) per core (based on the lscpu command) from the VM *infra-69* which only has 1 thread per core. We believe this accounts for the difference in CPU for small concurrent transactions in the OLTP workload. Each of the databases had a memory *(SGA)* configuration of 3.5Gb, if the SQL statement executed in the workload requires a memory larger than 3.5Gb, which is more common in OLAP and Data Mart workloads then sorts will go to disk. Database memory configurations influence the database execution plans and optimisers and this sensitivity is reflected in the IOPS's charts shown in Fig's 3b, 4b and 5b.



(a) Host Total IO's Made 72 hours (a) Host CPU Load Avg (15Mins) 72 hours (b) Host CPU Utilisation 72 Hours

Figure 6: Results HOST Metrics OLTP: workload patterns for the 72 hour period.

4.6 Experiment Two - Single Instance Standby Configurations

The Second set of experiments was to introduce a more complicated environment executing one workload *(OLTP)* on a single instance *primary* database with a physical standby database kept in sync using the Data Guard technology (Oracle Data Guard [18]) across the two sites, as shown in Fig 2b. A key factor in this experiment is that the physical standby database is always in a recovering state and therefore is not opened to accept SQL connections in the same way as a normal *(primary)* database. Therefore the agent is unable to gather the instance based metrics, so we capture host based metrics to compare and contrast the workload:

- CPU load over 15mins - This is the output from the *"Top"* command executed in linux, this measurement is a number using or waiting for CPU resources. For example if there is a 1, then on average 1 process over the 15 min time period is using or waiting for CPU.
- CPU Utilisation Percentage - This is based on the *"MPSTAT -P ALL"* command and looks at the percentage of all cpu's being used .
- TotalIOSMade - This is the total physical reads and total physical writes per 15 minute interval on the host.
- MaxIOSperSec - This is the Maximum physical reads and physical writes per sec.

The two VM's are located within the same site but in different rooms, Data Guard is configured using Maximum Performance mode to allow for network drops in the connectivity between the two physical locations. The database configurations were the same in Instance Parameters, Software Version and Patch Level. The Hardware configurations were the same in OS Level, Kernel Version and memory configuration. We capture the metrics of each workload and analyse the consistency of the metrics, as shown graphically in Figure 6.

4.7 Results and Analysis Experiment Two - OLTP Workload

The results for OLTP covering CPU and IOPS/s are shown graphically in Figure 6. Relying on host based metrics has a profound effect in the ability to compare and contrast different CPU models, as there is no common denominator (SPECInt) calculated. It also becomes difficult if there are multiple standby databases existing in the same environment. When the workloads were compared between the hosts, due to the nature of the physical standby and the primary behaving, as designed, in a completely different way, the graphs clearly show that the standby database has a considerably lower utilisation of CPU and IO resources. This is for several reasons:

- A physical Standby Database is in recovery mode therefore is not open for SQL DML or DDL in the same manner as a primary database is opened in normal mode. Therefore processes are not spawned at OS level/Database level, consuming resources such as Memory, CPU.
- A Physical standby applies *"Archivelogs"* and therefore is much more dependent on Physical Writes as these logs *(changes)* are applied on the standby from the primary database, therefore less IO load is generated.
- The reduction in IOPS/s is also attributed to DML/DDL is not being executed on the standby database in the same manner as a primary database (e.g. rows are not being returned as part of a query result set).

It was clear after the first experiment *OLTP*, that the workloads would be profoundly different in their footprint regardless of the workload being executed, so we have not included the results of the other workloads namely, OLAP and Data Mart.

4.8 Experiment Three - Clustered Database (Advanced Configuration)

The final set of experiments was to execute three the workloads on a more advanced configuration, a two-node clustered database running in an Engineered system (Exadata X5-2 platform) [1], illustrated in Fig 2. During the experiment, compute nodes are closed down to simulate a fail-over. The database configurations were the same in Instance Parameters, Software Version and Patch Level. The hardware configurations were the same in OS Level, Kernel Version and memory configuration. A difference in this experiment from the previous two is that the physical hardware and database are clustered. In this experiment we leverage the Exadata Technology in the IO substructure.

4.9 Results and Analysis Experiment Three - OLTP Workload

The results for OLTP covering Memory, CPU and IOPS/s are shown graphically in Fig 7. The OLTP workload was amended to run from node 1 for the second 24 hours and this is reflected in all three of the graphs, when the instance DBM012 is very much busier than instance DBM011. The workloads are then spread evenly for the following 48 hours.

- *CPU utilisation* - for the first 24 hours, the workloads were executed fairly evenly across the cluster with a workload of 2000 users connecting consistently with peaks of 1000 users at peak times, and the CPU showed similar patterns during the workload execution.

- *CPU utilisation* - When the workload ran abnormally and all users (3000 users) ran from one node, in the second 24 hours, then the CPU utilisation did almost double in usage as expected. The increase was approximately +99% (Day 7 Hour 15)

- *IOPS/s* - The IOPS's utilisation for the first 24 hours was similar, as expected, when the workloads were evenly spread. However when the workloads were run from node 2 in the second 24 hours the IOPS increase significantly, as expected. The IOPS during the failure period was as expected, an increase of +99% (Day 7 Hour 15).

- *IOPS/S Spike* - there are two major spikes occurring at Day 7 Hour 2 and Day 8 Hour 2, these are Level 0 database backups than only run from node 1 (DBM011)

- *Memory Consumption* - The maximum memory utilisation across both instances was consistent during the first 24 hours when the workload was evenly spread. The memory configuration on DBM012 is sufficient to handle the 3000 users during the failover period, although the increase in memory used on DBM012 was only +45%

In general, the conclusion from this experiment when executing the OLTP workloads was, it cannot be assumed that when a workload fails over from one node (database instance) to another node (database instance) the footprint will be double in terms of Memory. The workload did double for CPU and IOPS/s. The results show there is an increase in IOPS/s, Memory and CPU. The difference during normal running conditions (i.e. when workloads are evenly spread) was the following: +31% (Day 7 Hour 3) CPU, +2% Memory (Day 6 Hour 21) and +1% (Day 6 Hour 12) IOPS. When the workload failed over there was a difference of +97% (Day 7 Hour 9) CPU, +99% (Day 7 Hour 20) Memory and +99% (Day 08 Hour 10) IOPS. There are two large spikes at Day 7 Hour 2 and Day 8 Hour 2; these are Level 0 RMAN backups which account for the large IOPS readings. The database instance was sufficiently sized to handle both workloads otherwise we would of expected to see out of memory errors in the database instance alert file.

4.10 Results and Analysis Experiment Three - OLAP Workload

The results OLAP covering Memory, CPU and IOPS/s are shown graphically in Fig 8. The OLAP workload was amended to run from node 1 for the first 24 hours and this is clearly reflected in all three of the graphs, as the instance DBM011 is very much busier than instance DBM012 during this period. The workloads are then spread evenly for the following 48 hours.

- *CPU utilisation* - for the first 24 hours, node 1 ran the whole workload of 400 users and thus the DBM011 instance is busier compared with the workload across days two and three; as expected, utilization is effectively doubled, at +99%.

- *CPU utilisation* - when the workload ran normally (400 users) across both nodes then the utilisation was similar in its SPECint count with a difference of approximately +20%.

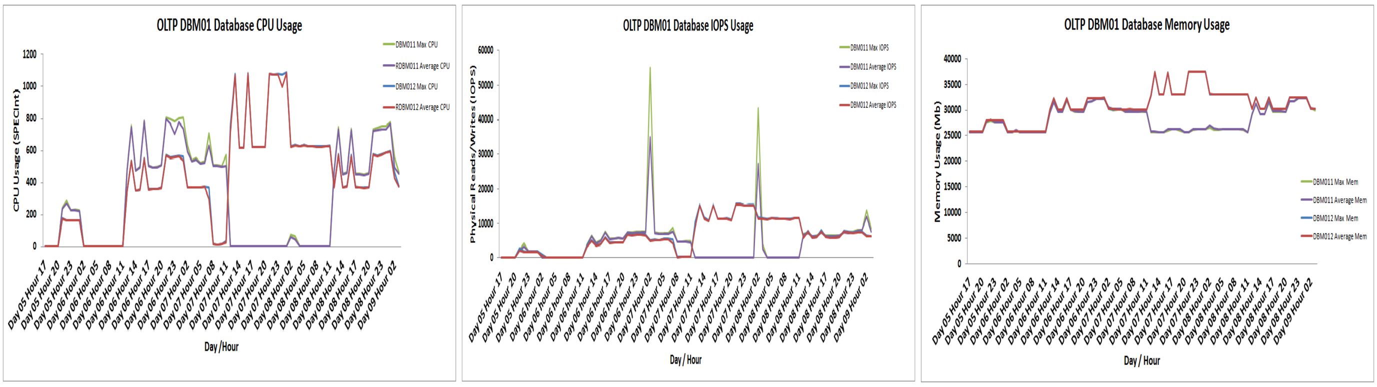
- *IOPS/s* - The IOPS's utilisation for the first 24 hours was busier on node 1, as expected, than node 2 given that both workloads were executed from DBM011 instance. The IOPS utilisation was almost double +99% (Day 25 Hour 05) the amount from the second period of time (Day 26 Hour 05) when the workloads were spread evenly across both instances.

- *Memory Consumption* - The maximum memory utilisation observed across both instances was consistent with the workload, the first 24 hours when the workload ran from node 1 is as expected in that there was sufficient memory to serve both workloads. However there is a difference of +55% (Day 25 Hour 04) in memory between nodes 1 and 2. For the second 24 hours, as the workloads reverted back to their normal hosts I.E. spread evenly across both nodes, their utilisation is similar with a difference of +1% (Day 26 Hour 04) between the nodes in memory utilisation.

In general, the conclusion from this experiment when executing the OLAP workloads was that it cannot be assumed that when a workload fails over from one node (database instance) to another node the footprint will be double in terms of Memory. For the metrics IOPS and CPU the increase was almost double; CPU had a difference of +99% (Day 25 Hour 04) and IOPS +99% (Day 24 Hour 04). When the workload was spread evenly across both nodes the differences between the nodes where CPU +20% (Day 26 Hour 3), Memory +2% (Day 26 Hour 3) and IOPS +1% (Day 26 Hour 4). The database instance was sufficiently sized to handle both workloads otherwise we would of expected to see out of memory errors in the database instance alert file.

4.11 Results and Analysis Experiment Three - Data Mart Workload

The results are as follows for the Data Mart workloads covering Memory, CPU and IOPS/s, as shown graphically in Fig 9. The Data Mart workload was run normally for the first 24 hours, which is reflected in the workloads being similar for this period. A simulated failure of database instance DBM011 is then performed and all connections then failover to DBM012 on node 2 for the second 24 hours. This is

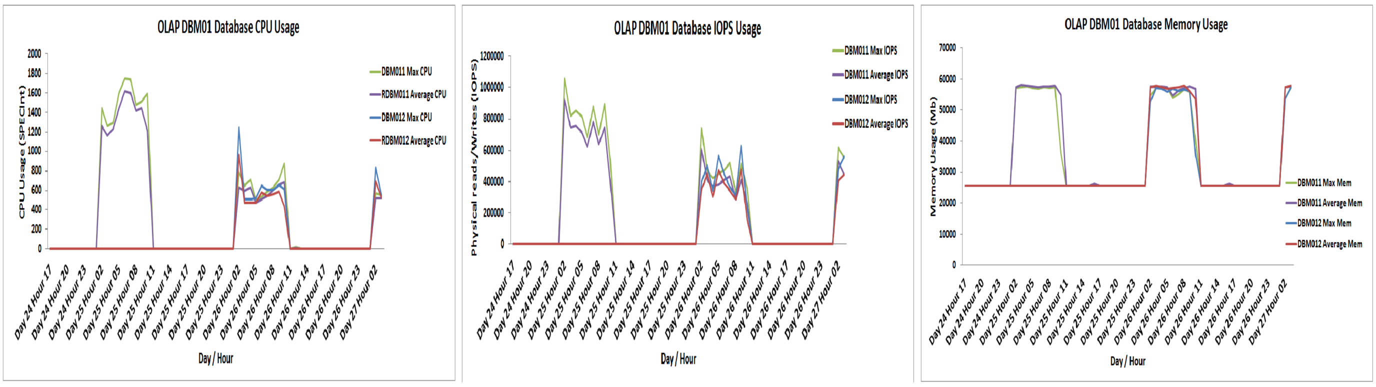


(a) CPU 72 hours

(b) IOPS/s 72 Hours

(c) Memory 72 Hours

Figure 7: Results RAC OLTP: workload patterns for the 72 hour period.

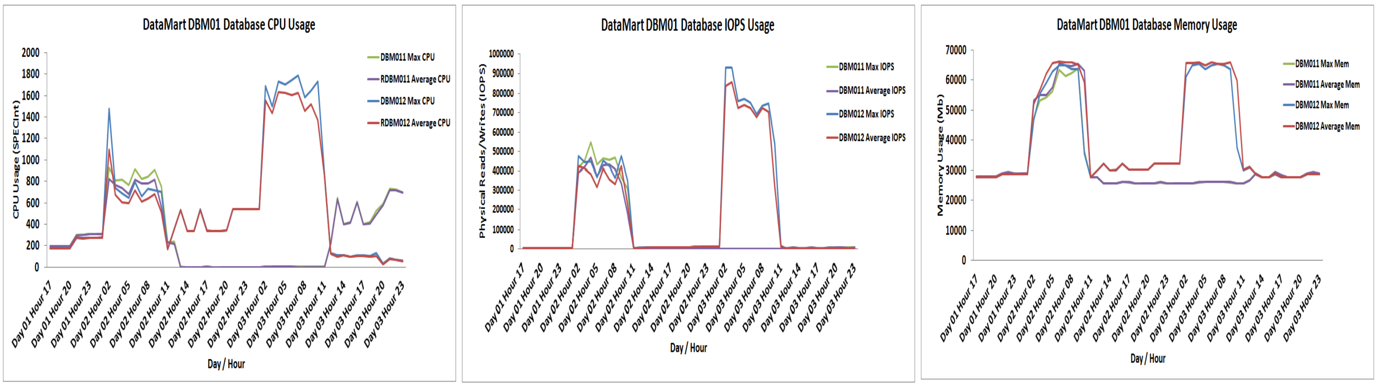


(a) CPU 72 hours

(b) IOPS/s 72 Hours

(c) Memory 72 Hours

Figure 8: Results RAC OLAP: workload patterns for the 72 hour period.



(a) CPU 72 hours

(b) IOPS/s 72 Hours

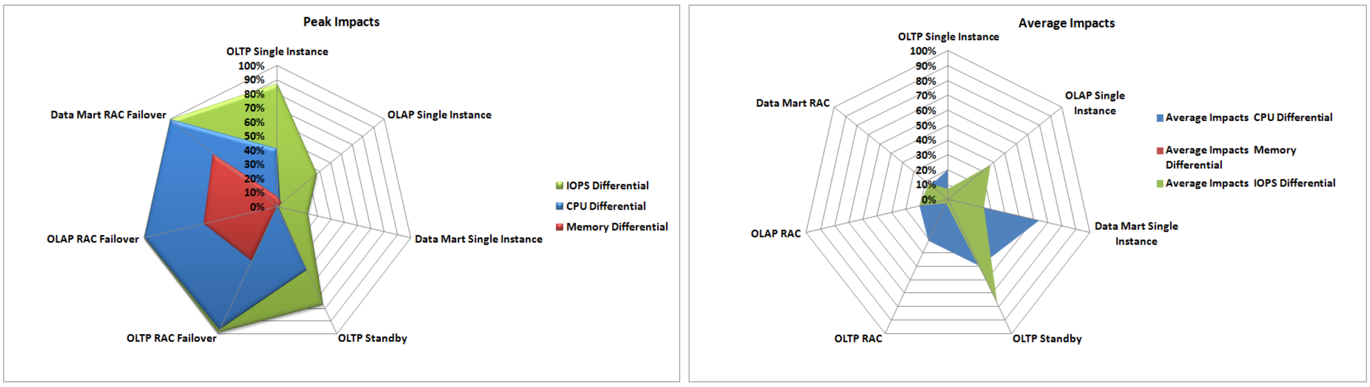
(c) Memory 72 Hours

Figure 9: Results RAC Data Mart: workload patterns for the 72 hour period.

reflected in all three of the graphs as the instance DBM012 becomes much busier than instance DBM011.

- *CPU utilisation* - For the first 24 hours, the workloads were executed fairly evenly across the cluster with a workload of 2700 users connecting at different times from the two nodes and the SPECInt count was similar with a average CPU difference of +15% (Day 2 Hour 04).

- *CPU utilisation* - When the workload ran abnormally and all users (2700 users) ran from one node, in the second 24 hours, then the CPU utilisation almost doubled in usage as expected +99% (Day 3 Hour 04).



(a) Volatility of workload Peak

(b) Volatility of workload Avg

Figure 10: Workload Impacts

- *IOPS/s* - The IOPS's utilisation for the first 24 hours was similar, as expected, when the workloads were evenly spread with a difference on average of +17% (Day 2 Hour 04). However, when the workloads were run from node 2 in the second 24 hours the IOPS increased significantly, rising to almost double at +99% (Day 3 Hour 04).

- *Memory Consumption* - The maximum memory utilisation across both instances was as expected during the first 24 hours, when the workloads were evenly spread, showing a difference of +9% (Day 2 Hour 04). This behaviour was not expected during the failover period when all users execute their workload on DBM012 as the utilisation difference is +60% (day 3 Hour 04). The memory configuration on DBM012 is sufficient to handle the 2700 users.

In general, the conclusion from this experiment when executing the Data Mart workloads was, it cannot be assumed that when a workload fails over from one node (database instance) to another node the footprint will be double in terms of memory, as it only increased by approximately +60%. CPU and IOPS however, did double in its usage to approximately +99%. When the workload was spread evenly the average utilisation had a difference of CPU +15% (Day 2 Hour 04), Memory +9% (Day 2 Hour 04) and IOPS +17% (Day 2 Hour 04).

5. CONCLUSIONS AND FUTURE WORK

From the experiments conducted and the model we proposed, we conclude that capacity planning of databases that employ advanced configurations such as Clustering and Standby Databases is not a simple exercise. Taking the Average and Maximum readings for each metric (CPU, Memory Utilisation and IOPS) over a period of 72 hours, the outputs are volatile. One should not assume that a workload running on one database instance configured in one type of system will consume the same amount of resource as an another database instance running on another system, regardless of similarity; this is clearly shown in Fig 10 (a) (OLTP, OLAP, Data Mart RAC Failovers). These charts show us that as workloads become assimilated they completely change as the difference grows, sometimes considerably. The differences between the footprints based on configuration can vary between +10% (CPU OLAP RAC) in normal circumstances shown in Fig 10 (b) to 99% (CPU OLAP RAC) as shown in Fig 10 (a). Fig 10(a & b, OLTP Standby) also highlights that configuration has a big impact on capacity planning databases with advanced configurations, such as standby databases.

In this paper we highlighted the problems that organisations are faced with *over-estimation* and *under-estimation* when trying to budget on non-cloud compliant financial models such as capex or cloud compliant models, which are subscription based. Accurate capacity planning can help in reducing wastage when metrics are captured and the assumption of workloads being the same is not employed. Capturing and storing the data in a central repository, like the approach we proposed, allowed us to mine the data successfully without the labour intensive analysis that often accompanies a capacity planning exercise.

The main points from this work are.

1. When capacity planning DBaaS, it should be done on a *instance-by-instance* basis and not at a database level - this is especially the case in clustered environments where workloads can move between one database and another or fail-over technology is employed.

2. Metrics need to be captured at different layers of the infrastructure in advanced configurations, for example in the storage layer, caching can mask IOPS causing the workload to behave differently.

3. Hypervisors and VMManagers can influence capacity planning as these tools allocate resource. For example, a CPU can be dissected and allocated as a vcpu (Oracle VM) [2]. How does one know that the CPU assigned is a full CPU? The Oracle Software and the database itself may assume that a full CPU was made available, when in fact it was assigned 0.9 of a CPU due to overheads.

4. CPU configuration *(Thread(s) per core)* within a VM has a profound effect when capacity planning. We observed in experiment one (OLTP and Data Mart) that small concurrent transactions in the OLTP workload executed on VM acs-163 were a lot more efficient than

the same workload executed on another VM with lower thread(s) per core, and this is reflected in Figures 3, 4 and 5.

5. SPECInt benchmark is a valid benchmark when comparing one varient of CPU with another, especially when trying to capacity plan databases with a view to a migration or upgrade of the infrastructure.

6. Standby Databases presented a different footprint. A standby database is always in a *mounted* state and therefore is configured in a recovering mode by applying logs or changes from the primary. It should not be assumed that the footprints are the same.

7. In environments that employ standby database configurations, metrics that are available for collection on the primary database are not available on the standby, namely physical reads/writes, CPU and memory, thus gathering accurate metrics is impractical. Metrics can be gathered at a host level, however if multiple standby databases are running on the same host this makes reconciliation of which database is using what more challenging.

8. In environments that employ clustered databases, if a workload running on one node fails-over from another node within the cluster, one should not assume that the properties of the composed workload will follow obviously from its constituents. Upon failover, the workload from the failing node is assimilated, with the result being the formation of a completely new footprint.

Future work is to conduct the same type of experiments between different database versions, for example a workload running on Oracle Database Version 10G/11G and Oracle Database Version 12C, analysing if the internal database algorithms have any influence and by how much. However techniques already exist that go some way to answering this question through the use of a product called Database replay [7]. Being able to gather metrics from a standby database instance for CPU, IOPS and Memory is critical for our model as this would allow us to accurately analyse the CPU such as SPECInt, Memory and IOPS's. We could configure a custom metric to execute internal queries against the standby database, and this is now in the design phase, but until then capacity planning architectures with standby database will need to rely on host metrics.

6. REFERENCES

[1] U. Author. Oracle exadata database machine. *Security Overview*, 2011.

[2] O. Corporation. Oracle vm concept guide for release 3.3.

[3] T. P. P. Council. Tpc-ds benchmark.

[4] T. P. P. Council. Tpc benchmark h standard specification version 2.1. 0, 2003.

[5] T. P. P. Council. Tpc benchmark c (standard specification, revision 5.11), 2010. *URL: http://www. tpc. org/tpcc*, 2010.

[6] K. M. Dixit. Overview of the spec benchmarks., 1993.

[7] L. Galanis, S. Buranawatanachoke, R. Colle, B. Dageville, K. Dias, J. Klein, S. Papadomanolakis, L. L. Tan, V. Venkataramani, Y. Wang, and G. Wood. Oracle database replay. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1159–1170, New York, NY, USA, 2008. ACM.

[8] D. Giles. Swingbench 2.2 reference and user guide.

[9] M. Guidolin, S. Hyde, D. McMillan, and S. Ono. Non-linear predictability in stock and bond returns: When and where is it exploitable? *International journal of forecasting*, 25(2):373–399, 2009.

[10] H. Hacıgümüş, J. Tatemura, Y. Chi, W. Hsiung, H. Jafarpour, H. Moon, and O. Po. Clouddb: A data store for all sizes in the cloud. *Internet: http://www. neclabs. com/dm/CloudDBweb. Pdf,[January 25, 2012]*, 2012.

[11] Y. Kouki and T. Ledoux. Sla-driven capacity planning for cloud applications. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 135–140. IEEE, 2012.

[12] C. Loboz. Cloud resource usageĂŤtailed distributions invalidating traditional capacity planning models. *Journal of grid computing*, 10(1):85–108, 2012.

[13] S. Mahambre, P. Kulkarni, U. Bellur, G. Chafle, and D. Deshpande. Workload characterization for capacity planning and performance management in iaas cloud. In *Cloud Computing in Emerging Markets (CCEM), 2012 IEEE International Conference on*, pages 1–7. IEEE, 2012.

[14] R. Moussa and H. Badir. Data warehouse systems in the cloud: Rise to the benchmarking challenge. *IJ Comput. Appl.*, 20(4):245–254, 2013.

[15] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and resource modeling in highly-concurrent oltp workloads. In *Proceedings of the 2013 acm sigmod international conference on management of data*, pages 301–312. ACM, 2013.

[16] B. Mozafari, C. Curino, and S. Madden. Dbseer: Resource and performance prediction for building a next generation database cloud. In *CIDR*, 2013.

[17] J. Murphy. Performance engineering for cloud computing. In *European Performance Engineering Workshop*, pages 1–9. Springer, 2011.

[18] A. Ray. Oracle data guard: Ensuring disaster recovery for the enterprise. *An Oracle white paper*, 2002.

[19] S. Sahri, R. Moussa, D. D. Long, and S. Benbernou. Dbaas-expert: A recommender for the selection of the right cloud database. In *International Symposium on Methodologies for Intelligent Systems*, pages 315–324. Springer, 2014.

[20] S. Shang, Y. Wu, J. Jiang, and W. Zheng. An intelligent capacity planning model for cloud market. *Journal of Internet Services and Information Security*, 1(1):37–45, 2011.

[21] T. Yu, J. Qiu, B. Reinwald, L. Zhi, Q. Wang, and N. Wang. Intelligent database placement in cloud environment. In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pages 544–551. IEEE, 2012.

Herding the elephants: Workload-level optimization strategies for Hadoop

Sandeep Akinapelli
Cloudera, Palo Alto, CA
sakinapelli@cloudera.com

Ravi Shetye
Cloudera, Palo Alto, CA
ravi@cloudera.com

Sangeeta T.
Cloudera, Palo Alto, CA
sangeeta@cloudera.com

ABSTRACT

With the growing maturity of SQL-on-Hadoop engines such as Hive, Impala, and Spark SQL, many enterprise customers are deploying new and legacy SQL applications on them to reduce costs and exploit the storage and computing power of large Hadoop clusters. On the enterprise data warehouse (EDW) front, customers want to reduce operational overhead of their legacy applications by processing portions of SQL workloads better suited to Hadoop on these SQL-on-Hadoop platforms - while retaining operational queries on their existing EDW systems. Once they identify the SQL queries to offload, deploying them to Hadoop as-is may not be prudent or even possible, given the disparities in the underlying architectures and the different levels of SQL support on EDW and the SQL-on-Hadoop platforms. The scale at which these SQL applications operate on Hadoop is sometimes factors larger than what traditional relational databases handle, calling for new workload level analytics mechanisms, optimized data models and in some instances query rewrites in order to best exploit Hadoop.

An example is aggregate tables (also known as materialized tables) that reporting and analytical workloads heavily depend on. These tables need to be crafted carefully to benefit significant portions of the SQL workload. Another is the handling of UPDATEs - in ETL workloads where a table may require updating; or in slowly changing dimension tables. Both these SQL features are not fully supported and hence have been underutilized in the Hadoop context, largely because UPDATEs are difficult to support given the immutable properties of the underlying HDFS.

In this paper we elaborate on techniques to take advantage of these important SQL features at scale. First, we propose extensions and optimizations to scale existing techniques that discover the most appropriate aggregate tables to create. Our approach uses advanced analytics over SQL queries in an entire workload to identify clusters of similar queries; each cluster then serves as a targeted query set for discovering the best-suited aggregate tables. We compare the performance and quality of the aggregate tables created with and without this clustering approach. Next, we describe an algorithm to consolidate similar UPDATEs together to reduce the number of UPDATEs to be applied to a given table.

While our implementation is discussed in the context of Hadoop, the underlying concepts are generic and can be adopted by EDW and BI systems to optimize aggregate table creation and consolidate UPDATEs.

CCS Concepts

•**Information systems → Database utilities and tools; Relational database model;**

Keywords

Query optimization; Hadoop; Hive; Impala; BI reporting

1. INTRODUCTION

Large customer deployments on Hadoop often include several thousand tables many of which are very wide. For example, in the retail sector, we have observed customer workloads that issue over 500K queries a day over a million tables some of which have 50,000 columns. Many of these queries share some common clustering characteristics; i.e. in a BI or reporting workload we may find clusters of queries that perform highly similar operations on a common set of columns over a common set of tables. Or in an ETL workload, UPDATEs on a certain set of columns over a common set of tables may be highly prevalent. But at such large scales, detecting common characteristics, identifying the set of queries that exhibit these characteristics and using this knowledge to choose the right data models to optimize these queries is a challenging task. Automated workload level optimization strategies that analyze these large volumes of queries and offer the most relevant optimization recommendations can go a long way in easing this task. Thus for the BI or reporting workload, creating a set of aggregate tables that benefit performance of a set of queries is a useful recommendation; while for the ETL case detecting UPDATEs that can be consolidated together can help overall performance of the UPDATEs.

Aggregate tables are an important feature for Business Intelligence (BI) workloads and various forms of it are supported by many EDW vendors including Oracle [8], Microsoft SQL Server [7] and IBM DB2 [15] as well as BI tools such as Microstrategy [13] and IBM Cognos [9]. Here, data required by several different user or application queries are joined and aggregated apriori and materialized into an aggregate table. Reporting and analytic queries then query these aggregate tables, which reduces processing time during query execution resulting in improved performance of the queries. This is an example aggregate table over the TPC-H workload schema:

```
CREATE TABLE aggtable_888026409 AS
 SELECT lineitem.l_quantity
  , lineitem.l_discount
  , lineitem.l_shipinstruct
  , lineitem.l_commitdate
  , lineitem.l_shipmode
  , orders.o_orderpriority
  , orders.o_orderdate
  , orders.o_orderstatus
  , supplier.s_name
  , supplier.s_comment
  , Sum (orders.o_totalprice)
  , Sum (lineitem.l_extendedprice)
 FROM   lineitem
  , orders
  , supplier
 WHERE  lineitem.l_orderkey = orders.o_orderkey
  AND lineitem.l_suppkey = supplier.s_suppkey
 GROUP  BY lineitem.l_quantity
  , lineitem.l_discount
  , lineitem.l_shipinstruct
  , lineitem.l_commitdate
  , lineitem.l_shipmode
  , orders.o_orderdate
  , orders.o_orderpriority
  , orders.o_orderstatus
  , supplier.s_name
  , supplier.s_comment
```

The aggregate table above can be used to answer queries which refer the same set of tables(or more), joined on same condition and refer columns which are projected in aggregated table. Few sample queries which can benefit from the above aggregate table are:

```
SELECT Concat(supplier.s_name,
 orders.o_orderdate) supp_namedate
, lineitem.l_quantity
, lineitem.l_discount
, Sum(lineitem.l_extendedprice) sum_price
, Sum(orders.o_totalprice)      total_price
FROM   lineitem
 JOIN  part
  ON ( lineitem.l_partkey = part.p_partkey )
 JOIN  orders
  ON ( lineitem.l_orderkey = orders.o_orderkey )
 JOIN  supplier
  ON ( lineitem.l_suppkey = supplier.s_suppkey )
WHERE  lineitem.l_quantity BETWEEN 10 AND 150
 AND lineitem.l_shipinstruct <> ‘deliver IN person’
 AND lineitem.commitdate BETWEEN ‘11/01/2014’
         AND ‘11/30/2014’
 AND lineitem.l_shipmode NOT IN (‘AIR’, ‘air reg’)
 AND orders.o_orderpriority IN (‘1-URGENT’, ‘2-high’)
GROUP BY Concat(supplier.s_name, orders.o_orderdate)
 , lineitem.l_quantity
 , lineitem.l_discount
```

or

```
SELECT lineitem.l_shipmode
 , Sum(orders.o_totalprice)
 , Sum (lineitem.l_extendedprice)
FROM lineitem
 JOIN orders
  ON ( lineitem.l_orderkey = orders.o_orderkey )
 JOIN supplier
  ON ( lineitem.l_suppkey = supplier.s_suppkey )
WHERE ( lineitem.l_quantity BETWEEN 10 AND 150
 AND lineitem.l_shipinstruct <> ‘DELIVER IN PERSON’
 AND lineitem.commitdate BETWEEN
 ‘11/01/2014’ AND ‘11/30/2014’
 AND supplier.s_comment Like ‘\%customer\%complaints\%’
 AND orders.o_orderstatus =‘f’
GROUP BY
  lineitem.l_shipmode
```

Similarly, UPDATE statements in various flavors that modify rows in a table via a direct UPDATE or with the query results from another query, have been supported in relational DBMS offerings for several decades. In some scenarios, consolidating UPDATEs can produce performance benefits. For example combining the following two simple statements:

```
UPDATE customer
 SET   customer.email_id=‘bob.johnson@edbt.org’
WHERE  customer.firstname=‘Bob’
 AND   customer.last_name=‘Johnson’

UPDATE customer
 SET   customer.organization=‘Engineering’
WHERE  customer.firstname=‘Bob’
 AND   customer.last_name=‘Johnson’
```

into a single UPDATE statement as follows:

```
UPDATE customer
 SET   customer.email_id=‘bob.johnson@edbt.org’,
       customer.organization=‘Engineering’
WHERE  customer.firstname=‘Bob’
 AND   customer.last_name=‘Johnson’
```

Such consolidation reduces the number of UPDATE queries on the source table ‘customer’ and minimizes the I/O on the table.

Existing commercial offerings also support the ‘REFRESH’ option to propagate changes to aggregate tables whenever the underlying source tables are updated. Generally, this requires a mechanism to UPDATE rows in the aggregate tables. However, the immutable properties of HDFS in Hadoop, which is highly optimized for write-once-read-many data operations, poses problems for implementing the ‘REFRESH’ option in Hive and Impala. This hampers developing functionality for UPDATE statements - which has largely lead Hadoop-based SQL vendors to shy away from or offer limited support for UPDATE-related features.

Based on our learnings from several customer engagements, some important observations surface:

1. In Hadoop, highly parallelized processing and optimized execution engines on systems such as Hive and Impala enable rebuilding aggregate tables from scratch very quickly, making UPDATEs unnecessary and mitigating the HDFS related immutability issues in many EDW workloads.

2. Many aggregate tables are temporal in nature. For example, quarterly financial reports that require data from only three months, in which case the aggregate tables that feed these reports can be data partitioned on a monthly basis on Hive and Impala. Smaller portions of giant source tables need to be queried to populate these aggregate tables. Only the impacted partitions of the aggregate tables need to be written, making modifications to aggregate tables less expensive. Hence, instead of using UPDATES to modify them, new time-based partitions (by month or day) can be added and older ones discarded. SQL constructs such as INSERT with OVERWRITE supported on Hive and Impala, can be used to mimic this REFRESH functionality. And SQL views can be used to allow easy switching between an older and newer version of the same data.

3. With the introduction of new Hadoop features such as the Apache Kudu integration [12], a viable alternative to using HDFS is now available. Hence UPDATEs can now be supported for certain workloads.

UPDATE statements used to perform tasks such as address cleanup in ETL workloads or modify slowly-changing dimension tables in BI and Analytic workloads are different in nature from highly concurrent OLTP style UPDATEs present in traditional operational systems. In this case, UPDATEs are concentrated on certain tables and are less frequent. If the temporal nature of data mentioned above can be exploited, partitioning techniques to mimic UPDATEs are possible as are some other SQL join-based techniques discussed later in this paper.

Given the importance of these two SQL features, BI Users and Hadoop developers are adopting one of the above mentioned strategies; and require recommendations on which aggregate tables to create, and how to consolidate UPDATE statements, to optimize the performance of their queries on Hadoop. In the following sections, we describe our algorithm for aggregate table creation. And compare the efficiency and quality of the aggregate tables generated when the input to the algorithm is all queries in a workload versus targeted sets of highly similar queries derived from the workload. A clustering algorithm performs advanced analytics over all the queries in a workload, to extract these highly similar query sets. We also discuss techniques and algorithms for consolidation of UPDATE statements, prior to applying them on Hadoop.

2. BACKGROUND AND RELATED WORK

Aggregate table advisors are available in several commercial EDW and BI offerings. In some of these offerings, the onus is on the user to provide a representative workload - i.e. a sample set of queries to use for deriving aggregate tables. Others require query execution plans to provide recommendations. The DB2 Design Advisor in [15] discusses the issue of reducing the size of the sample workload to reduce the search space for aggregate table recommendations, while the Microsoft paper [3] details specific mechanisms to compress SQL workloads. Our approach takes a SQL query log as an input workload (all queries executed over a period of time in a EDW system) and identifies semantically unique queries discarding duplicates. We use the structure of the SQL query when identifying the duplicates which means the changes in the literal values result in identifying these queries as duplicates. Advanced analytics are then deployed on the SQL structures of these semantically unique queries to discover clusters of similar query sets. This enables quicker and more relevant aggregate table definitions because the set of queries that serve as input to aggregate table recommendations are highly similar.

After aggregate tables are set up, some DBMS and BI tools offerings are further capable of rewriting queries internally to use aggregate tables versus the base tables to optimize performance of queries. This feature also known as materialized views is not addressed in our paper. An example Hadoop implementation of materialized views is described in [14]. In our experience, BI tools are frequently used in reporting and analytic workloads deployed atop Hadoop and necessarily support materialized views. Hence we provide recommendations and the DDL definitions for the aggregate tables that users can create, using the BI tools of their choice.

On the Hadoop side, the [5] features revolve around using materialized views to better exploit Hadoop clusters. And in the Hive community, explicit support for materialized views is under development [10]. Again, these efforts are orthogonal to the aggregate table recommendations we provide. [11] seeks to solve the HDFS immutability issue and lift UPDATE restrictions. The techniques we propose in this paper are orthogonal and applicable at the SQL level - and seek to boost performance of SQL queries on Hadoop. Thus they can benefit both HDFS and Kudu-based Hadoop deployments.

3. THE SYSTEM

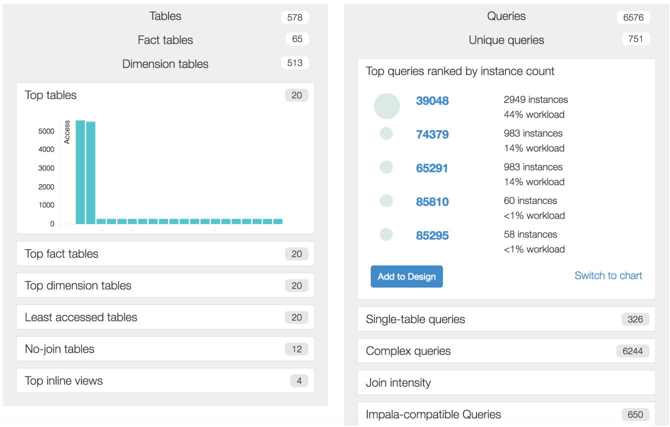
Our system is a workload-level optimization tool that analyzes SQL queries (from many popular RDBMS vendors) from sources such as query logs. It breaks down the individual SQL constructs in these queries and employs advanced analytics to

- identify semantically unique queries, thus eliminating duplicates
- discover top tables and queries in a workload as shown in Figure 1
- surface popular patterns like joins, filters and other SQL constructs used in the workload.

This analysis is further used to alert users to SQL syntax compatibility issues and other potential risks such as many-table joins that these queries could encounter on Hive or Impala providing recommendations on data model changes and query rewrites that can benefit performance of the queries on Hadoop.

The tool operates directly on SQL queries so does not require access to the underlying data in tables or to the Hadoop clusters that the workload may be deployed on. However, information such as the elapsed time for a query and statistics such as table volumes and number of distinct values (NDV) in columns, help improve the quality of our recommendations.

The recommendations include candidates for partitioning keys, denormalization, inline view materialization, aggregate tables and update consolidation. The last two recommendations are the focus of this paper.



Tables 578

Fact tables 65

Dimension tables 513

Top tables 20

Top fact tables 20

Top dimension tables 20

Least accessed tables 20

No-join tables 12

Top inline views 4

Queries 6576

Unique queries 751

Top queries ranked by instance count

39048 2949 instances 44% workload

74379 983 instances 14% workload

65291 983 instances 14% workload

85810 60 instances <1% workload

85295 58 instances <1% workload

Single-table queries 326

Complex queries 6244

Join intensity

Impala-compatible Queries 650

Figure 1: Workload Insights: Popular Queries and Patterns.

3.1 Aggregate Table Recommendation

Our algorithm to determine aggregate tables, is similar to [2] with a few important modifications, which are elaborated in the subsequent sections. The first step in determining the aggregate tables is to find a set of interesting table subsets. A table-subset T is interesting if materializing one or more views on T has the potential to reduce the cost of the workload significantly, i.e., above a given threshold.

In BI workloads, joins over 30 tables in a single query is not an infrequent scenario. Such workload characteristics could incur exponential costs while enumerating all interesting subsets. The enumeration of all interesting subsets of 30 tables is not practical hence we need a mechanism to reduce the overall number of interesting subsets. [1] presents efficient algorithms to enumerate all frequent sets and [4] presents a compact way of representing all the frequent sets. However generating aggregate tables on a subset of tables may be more beneficial than generating it over supersets. Since enumerating all subsets can be exponential, we need to select the subsets which are still a good representation of all the interesting subsets.

3.1.1 *Merge and Prune*

We address the problem of exponential subsets by constraining the size of the items at every step. During each step in subset formation, we merge some of the subsets early and then prune some of these subsets, without compromising on the quality of the output. We use the notations mentioned in Table 1 to describe the various concepts used in our mergeAndPrune algorithm. The detailed steps are outlined in the algorithm 1. The algorithm takes a set of sets of tables of a given size and returns the new set with some elements merged and removed from the input.

The metric TS-Cost(T) we use is the same as that mentioned in [2] which is the total cost of all queries in the workload where table-subset T occurs. After we enumerate all 2-subsets (subsets of size 2) we execute the algorithm in each step for merging and pruning the sets early. We start with a given element and collect the list of all candidates that it can be merged with. The merges are performed as long as the merged set is within a certain threshold. Experimental results indicated that a value of .85 to 0.95 is a good candidate for this threshold. We maintain a merge list and add the elements from the merge list to the prune list, only if there is no potential for the elements to form further combinations of tables.

Table 1: Notations used in mergeAndPrune

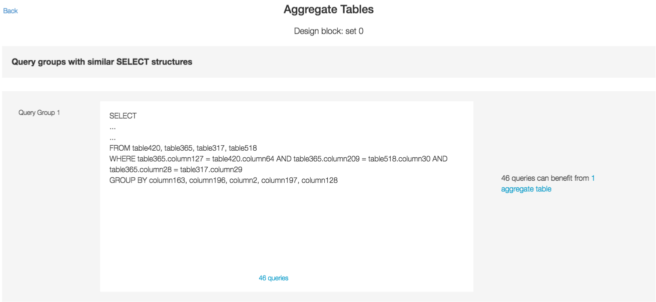
input	Set of sets of a given size formed by tables in the queries.
pruneSet	A subset of *input* that holds the list of elements that will be pruned from the *input* at the end of the iteration.
M	Holds the current table set that is considered for meging.
$MList$	Holds all the sets that can be merged with M.
mergedSets	Set of sets of tables that are formed by merging some elements from *input*.

Algorithm 1 Algorithm for merging and pruning interesting table subsets

```
function MERGEANDPRUNE
    for each $i \in$ input and $i \notin$ pruneSet do
        $M \leftarrow i$
        $MList \leftarrow \{i\}$
        for each $c \in$ input do
            if $c \subset M$ then
                $MList \leftarrow MList \cup c$
                continue
            end if
            ▷ determine if the merge item is effective and not too far off from the original
            if TS-COST(M ∪ c)/TS-COST(M) > MERGE_THRESHOLD then
                $M \leftarrow M \cup c$
                $MList \leftarrow MList \cup c$
            end if
        end for
        for each $m \in MList$ do
                ▷ retain candidates that we should not be pruning.
            if $\nexists$ s| s $\in$ input and s $\notin MList$ and s $\cap$ m $\neq \phi$ then
                ▷ find the candidates for pruning from input in the later step.
                $pruneSet \leftarrow pruneSet \cup m$
            end if
        end for
        $mergedSets \leftarrow mergedSets \cup M$
    end for
    $input \leftarrow input - pruneSet$
    return mergedSet
end function
```

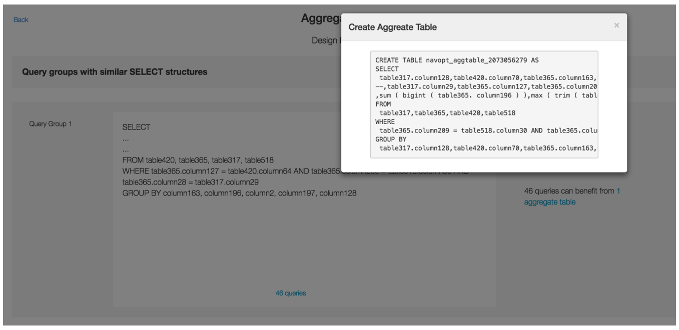
3.1.2 *Aggregate table creation using query clustering*

BI reporting workloads and analytical workloads typically generate queries against the same star/snowflake schema, but these queries select different sets of columns and invoke different sets of aggregate functions i.e SUM, COUNT etc. The clustering algorithm compares the similarity of each clause in the SQL query (i.e. SELECT list, FROM, WHERE, GROUPBY, etc.) to pull together highly similar


Back
Aggregate Tables
Design block: set 0
Query groups with similar SELECT structures
Query Group 1
46 queries can benefit from 1 aggregate table
46 queries

```
SELECT
...
...
FROM table420, table365, table317, table518
WHERE table365.column127 = table420.column64 AND table365.column209 = table518.column30 AND
table365.column28 = table317.column29
GROUP BY column163, column196, column2, column197, column128
```

Figure 2: Aggregate Table Candidate Queries


Back
Aggrega
Design
Query groups with similar SELECT structures
Query Group 1
Create Aggreate Table
×
46 queries can benefit from 1 aggregate table
46 queries

```
CREATE TABLE navopt_aggtable_2073056279 AS
SELECT
 table317.column128,table420.column70,table365.column163,
--,table317.column29,table365.column127,table365.column20
,sum ( bigint ( table365. column196 ) ),max ( trim ( tabl
FROM
 table317,table365,table420,table518
WHERE
 table365.column209 = table518.column30 AND table365.colu
GROUP BY
 table317.column128,table420.column70,table365.column163,
```

```
SELECT
...
...
FROM table420, table365, table317, table518
WHERE table365.column127 = table420.column64 AND table365
table365.column28 = table317.column29
GROUP BY column163, column196, column2, column197, column128
```

Figure 3: Aggregate Table DDL Generation

queries. Clustering queries in the workload based on the similarity of the SQL query structure collects together queries that access the same or almost similar table sets. Figure 2 and figure 3, depicts our implementation. Figure 2 shows the aggregate query that is beneficial to the given cluster of queries.The number of queries in the cluster are shown on the right side. As shown in figure 3, users can also generate the DDL that creates the specified aggregate table.

3.2 Update Consolidation

Several customers have legacy applications that encapsulate ETL logic in SQL stored procedures and SQL scripting languages (such as Oracle PL/SQL or Teradata BTEQ). Neither Hive nor Impala support stored procedures, so the individual queries in these stored procedures need to be executed on Hive/Impala. This ETL logic many times includes UPDATE queries.

In this section, we focus on such UPDATE queries, specifically the issue of converting a sequence of UPDATE queries in a workflow into a smaller set of UPDATE queries. We call this UPDATE consolidation.

Table 2: Notations used in update consolidation

Q_i	ith query in the sequence.
SOURCETABLES(Q_i)	All the tables that the query reads from.
TARGETTABLE(Q_i)	The table that is updated as part of the given INSERT/ UPDATE/ DELETE query.
C_i	consolidation set i containing one or more queries.
READCOLS(e)	Set of all the columns that are read by the given query e. For a consolidated set e, this is the union of all the columns belonging to every query in the set.
WRITECOLS(e)	Set of all the columns that will be written by the given query e. For a consolidated set e, this is the union of all the columns belonging to every query in the set.
TYPE(Q_i)	type of the UPDATE query, 1 if it is a single table UPDATE; and 2 if more than one table is referenced in the query.
TYPE(C)	UPDATE type of all the queries contained in the set. A set only contains queries of same type. Hence its a single value indicating the update type of all queries.
SETEXPREQUAL(Q_i, C)	returns true if the set expression in the UPDATE query Q_i is same as one of the set expression in consolidate set C all other columns except those in set expression are not write conflicted

Most UPDATE queries in ETL workflows are not complex and largely following the patterns like:

```
UPDATE employee emp
 SET   salary = salary * 1.1
WHERE  emp.title = ‘Engineer’;

UPDATE emp
 FROM  employee emp ,
       department dept
 SET   emp.deptid = dept.deptid
WHERE  emp.deptid = dept.deptid
 AND   dept.deptno = 1
 AND   emp.title = ‘Engineer’
 AND   emp.status = ‘active’;
```

We classify these UPDATE queries into two categories: Type 1 and Type 2 UPDATEs:

- Type 1 UPDATEs are single table UPDATE queries with an optional WHERE clause.
- Type 2 UPDATEs involve updates to a single table based on querying multiple tables.

This distinction between UPDATE queries is important, because Type 1 and Type 2 UPDATE queries can never be consolidated together. To execute UPDATE queries on Hadoop, the typical process is to use the CREATE-JOIN-RENAME conversion mechanism. The three steps of the CREATE-JOIN-RENAME conversion mechanism are:

1. Create a temporary table by converting the UPDATE query into CREATE+SELECT query, containing the primary key and updated columns of the target table.
2. Create a new table by performing a LEFT OUTER JOIN of the original table with the temporary table.Non

null values in the temporary table get priority over the original table.

3. Drop the original table and RENAME the newly created table to that of the original table.

If these 3 steps are needed to process each UPDATE, executing a sequence of UPDATEs can become very expensive if the steps are repeated for each UPDATE query individually. An efficient way of executing sequential UPDATEs on Hadoop is to first consolidate the UPDATEs into a smaller set of queries. However, it is very important to attempt consolidation only when we can guarantee that the end state of the data in the tables remains exactly the same with both approaches - i.e. when applying one UPDATE at a time versus a consolidated UPDATE. Therefore, the algorithm has to check for interleaved INSERT/UPDATE/DELETE queries, be mindful of transactional boundaries, etc. - and only perform consolidation when it is safe to do so.

Partitioned tables can be updated using the PARTITION OVERWRITE functionality. If the UPDATE statement contains a WHERE clause on the partitioning column, then we can convert the corresponding UPDATE query into an INSERT OVERWRITE query along with the required partition specification. If the query is modifying a selected subset of rows in the partition, we still have to follow the above approach to compute the new rows for the partition, including the modified rows. In this case too, since a join is involved, it is beneficial to look at consolidation options.

Another commonly used workaround to mitigate UPDATE issues is to use database views, i.e. users access data pointed to by a normal table or in the Hadoop context a partitioned table through a view. After UPDATEs to the table are propagated to Hadoop by adding a new partition that contains updated data to the existing table or re-building the entire table that now reflects UPDATEs, the view definition is changed to now point at the newly available data. This way users have access to the 'old' data till the point of the switch. A similar approach, (but for the compaction use case) is discussed here [6]. Even with this mechanism, consolidating updates to a particular partition or table prior to applying the updates, can minimize IO costs.

3.2.1 Update Consolidation Algorithm

We use the notations mentioned in Table 2 to describe the various concepts used in our UPDATE consolidation algorithm.

The $findConsolidatedSets$ algorithm to consolidate UPDATE queries is shown in Algorithm 4. The algorithm starts with an empty set and adds the first UPDATE query it finds into the current consolidation set. Then it checks subsequent queries to see if there are any potential conflicts with the group in hand. Query Q_i conflicts with Q_j if Q_j is either reading or writing a table that Q_i writes to. We use the procedure *isReadWriteConfict* in Algorithm 2 to determine the same.The UPDATE queries Q_i and Q_j that are reading from the same set of tables and writing to the same table can conflict if one of the queries is writing to a column, which the other query is reading from. We use the procedure *isColumnConflict* in Algorithm 3 to determine the conflict.

Algorithm 2 Procedure to detect conflicting queries

```
function isReadWriteConfict($e_1$,$e_2$)
    if targetTable($e_1$) $\cap$ sourceTables($e_2$) = $\phi$ && targetTable($e_2$) $\cap$ sourceTables($e_1$) = $\phi$ && targetTable($e_2$) $\cap$ targetTable($e_1$) = $\phi$ then
        return True
    else
        return False
    end if
end function
```

Algorithm 3 Procedure to detect conflicting read/write columns

```
function isColumnConflict($e_1$,$e_2$)
    if writeCols($e_1$) $\cap$ readCols($e_2$) = $\phi$ && writeCols($e_2$) $\cap$ readCols($e_1$) = $\phi$ && writeCols($e_2$) $\cap$ writeCols($e_1$) = $\phi$ then
        return True
    else
        return False
    end if
end function
```

When we encounter a conflicting query, we stop the consolidation process. When two UPDATE queries Q_i and Q_j are in sequence with no conflicting queries in between, they can be considered for consolidation. So we check Q_i and Q_j for compatibility. Q_i and Q_j can be consolidated into one group if all the following conditions are met:

1. Q_i and Q_j are of the same UPDATE types - i.e. either both are Type 1 or both are Type 2.

2. For Type 1 UPDATEs, the target table is the same for Q_i and Q_j and there are no columns of Q_i and Q_j that are write conflicted.

3. For Type 2 UPDATEs, the source and target tables are the same for Q_i and Q_j (along with same join predicate) and there are no columns of Q_i and Q_j that are write conflicted.

Finally, we maintain a visited flag with each UPDATE query so that if there are interleaved UPDATEs between totally different UPDATE queries in the same stored procedure, they can be considered for consolidation.

Once we have identified a group of all consolidated sets, the conversion to the equivalent CREATE-JOIN-RENAME queries follows these steps. Here, without loss of generality, we assume that all WHERE predicates are in Conjunctive Normal Form.

1. We convert each of the
   'SET <col> = <colexpression> WHERE <predicates>'
   into
   'CASE WHEN <predicates> THEN <colexpression> ELSE <col> END as <col>'

2. For queries with same SET expression and different WHERE predicates, we create an OR clause for each of the WHERE predicates in the CASE block.

3. We take the WHERE predicates of all the queries and combine them using disjunction with the OR operator. If there is a common subexpression among WHERE predicates, we promote the common subexpression outwards.

Algorithm 4 Procedure to find and consolidate queries

```
function findConsolidatedSets
    $C \leftarrow \{\}$    ▷ current consolidated set
    $Q \leftarrow setOfInputQueries$
    while ∃ update query with visited(q) = False do
        for i←1 to |Q| do
            if $Q_i \neq$ Update Query then    ▷ insert or delete query
                if ¬ isReadWriteConflict(C,$Q_i$) then    ▷ conclude the current consolidated set and start a new set
                    output ← output ∪ C
                end if
                visited($Q_i$)← True ; continue
            end if
            if |C| = 0 and $Q_i$ is Update Query and visited($Q_i$) = False then
                visited($Q_i$)← True ; continue
            end if
            if type($Q_i$) ≠ type($C$) then
                output ← output ∪ C
                if visited($Q_i$) = False then
                    C ← {$Q_i$}
                else
                    C ← $\phi$
                end if
                visited($Q_i$)← True ; continue
            end if
            if type($Q_i$) = 1 and type($C$) = 1 then    ▷ Type 1 : Single table update query
                if targetTable($Q_i$) = targetTable($C$) then
                    if isColumnConflict(C,$Q_i$) or setExprEqual($Q_i$,$C$) then
                        if visited($Q_i$) = False then
                            C ← C ∪ $Q_i$
                        end if
                    else
                        output ← output ∪ C
                        if visited($Q_i$) = False then
                            C ← {$Q_i$}
                        else
                            C ← $\phi$
                        end if
                    end if
                    visited($Q_i$)← True ; continue
                end if
            end if
            if type($Q_i$) = 2 and type($C$) = 2 then    ▷ Type 2 : Multi table update query
                if targetTable($Q_i$) = targetTable($C$) and sourceTable($Q_i$) = sourceTable($C$) then
                    if isColumnConflict(C,$Q_i$) or setExprEqual($Q_i$,$C$) then
                        if visited($Q_i$) = False then
                            C ← C ∪ $Q_i$
                        end if
                        visited($Q_i$)← True ; continue
                    end if
                end if
                if ¬ isReadWriteConflict(C,$Q_i$) then
                    output ← output ∪ C
                    if visited($Q_i$) = False then
                        C ← {$Q_i$}
                    else
                        C ← $\phi$
                    end if
                    visited($Q_i$)← True ; continue
                end if
            end if
        end for
    end while
    return output
end function
```

Here are some examples of consolidations. The following Type 1 UPDATE queries that modify the table 'lineitem' - with or without filtering conditions:

```
UPDATE lineitem
 SET   l_receiptdate = Date_add(l_commitdate, 1)

UPDATE lineitem
 SET   l_shipmode = concat(l_shipmode,'-usps'),
WHERE  l_shipmode = 'MAIL'

UPDATE lineitem
 SET   l_discount = 0.2
WHERE  l_quantity > 20
```

can be consolidated and converted into a CREATE-JOIN-RENAME flow as follows:

```
CREATE table lineitem_tmp AS
SELECT Date_add(l_commitdate, 1) AS l_receiptdate
 , CASE
    WHEN l_shipmode = 'MAIL'
    THEN concat(l_shipmode,'-usps')
    ELSE l_shipmode
   END AS l_shipmode
 , CASE
    WHEN l_quantity > 20
    THEN 0.2
    ELSE l_discount 0
   END AS l_discount
 , l_orderkey
 , l_linenumber
FROM lineitem;

CREATE TABLE lineitem_updated AS
 SELECT orig.l_orderkey
  , orig.l_linenumber
  , Nvl(tmp.l_receiptdate, orig.l_receiptdate)
     AS l_receiptdate
  , Nvl(tmp.l_shipmode, orig.l_shipmode)
     AS l_shipmode
  , Nvl(tmp.l_discount, orig.l_discount)
     AS l_discount
  , l_partkey, l_suppkey, l_quantity, l_extendedprice
  , l_tax, l_returnflag, l_linestatus, l_shipdate
  , l_commitdate, l_shipinstruct, l_comment
FROM  lineitem orig
 LEFT OUTER JOIN lineitem_tmp tmp
 -- lineitem table primary key
  ON ( orig.l_orderkey = tmp.l_orderkey
   AND orig.l_linenumber = tmp.l_linenumber )

DROP TABLE lineitem;

ALTER TABLE lineitem_updated RENAME TO lineitem;
```

As another example consider the following Type 2 UPDATE queries, that modify the 'lineitem' table based on the results of a join with the 'orders' table:

```
UPDATE lineitem
 FROM lineitem l
    , orders o
 SET l.l_tax = 0.1
WHERE l.l_orderkey = o.o_orderkey
 AND o.o_totalprice BETWEEN 0 AND 50000
 AND o.o_orderpriority = '2-HIGH'
 AND o.o_orderstatus = 'F';

UPDATE lineitem
 FROM lineitem l
    , orders o
SET l_shipmode = 'AIR'
WHERE l.l_orderkey = o.o_orderkey
 AND o.o_totalprice BETWEEN 50001 AND 100000
 AND o.o_orderpriority = '2-HIGH'
 AND o.o_orderstatus = 'F';
```

can be consolidated and converted into a CREATE-JOIN-RENAME flow as follows:

```
CREATE TABLE lineitem_tmp AS
 SELECT CASE
     WHEN o.o_totalprice BETWEEN 0 AND 50000
     THEN 0.1 ELSE l_tax END AS l_tax
  , CASE
     WHEN o.o_totalprice BETWEEN 50001 AND 100000
     THEN 'AIR' ELSE l_shipmode
    END AS l_shipmode
  , l_orderkey
  , l_linenumber
 FROM lineitem l
  , orders o
 WHERE l.l_orderkey = o.o_orderkey
  AND o.o_totalprice BETWEEN 0 and 100000
  AND o.o_orderpriority = '2-HIGH'
  AND o.o_orderstatus = 'F';


CREATE TABLE lineitem_updated AS
 SELECT orig.l_shipdate
  , orig.l_commitdate
  , orig.l_receiptdate
  , orig.l_orderkey
  , orig.l_partkey
  , orig.l_suppkey
  , orig.l_linenumber
  , orig.l_extendedprice
  , Nvl(tmp.l_tax, orig.l_tax)  AS l_tax
  , orig.l_returnflag
  , orig.l_linestatus
  , Nvl(tmp.l_shipmode, orig.l_shipmode) AS l_shipmode
  , orig.l_shipinstruct
  , orig.l_discount
  , orig.l_comment
 FROM lineitem orig
  LEFT OUTER JOIN lineitem_tmp tmp
   ON ( orig.l_orderkey = tmp.l_orderkey
    AND orig.l_linenumber = tmp.l_linenumber )

DROP TABLE lineitem;

ALTER TABLE lineitem_updated RENAME TO lineitem;
```

We also looked at the problem of constructing a control flow graph of the stored procedure and performed a static analysis on this graph. If the number of different flows are manageably finite, we can generate a consolidation sequence for each of the different flows independently thus enabling the user to script these flows independently. However we omit all the implementation details here, as it is beyond the scope of this paper.

4. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of the various recommendations discussed in the previous sections using two different workloads. The first workload is TPC-H at the 100 GB scale, which we call TPCH-100. Our second workload belongs to a customer in the financial sector. This customer has 578 tables with 3038 number of columns. The table sizes vary from 500 GB to 5TB. We call this workload CUST-1.

We have setup representative clusters to measure the performance of the system. This cluster has 21 nodes with 1 master and 20 data nodes. The data nodes are the AWS m3.xlarge kind, with 4 core vCpu, 2.6 GHZ, 15GB of main memory and 2 X 40GB SSD storage. In all the experiments 'time' refers to the wall clock time as reported by the executing Hive query. There are no other queries running on the system. For simplicity, we ignore the HDFS and other OS caches. The experiments presented should be interpreted as directional rather than exhaustive empirical validation.

4.1 Aggregate Table Recommendation

For aggregate table generation we ran our experiments on the CUST-1 setup.

4.1.1 Clustering similar queries

In our first set of experiments we evaluate the quality of aggregate tables generated with and without clustering similar queries together. We divided a workload with 6597 queries into set of clusters using the clustering algorithm, thus reducing the number of queries to a group of smaller workloads. We empirically show how this approach provides aggregate table recommendations with better run time benefits. The first four smaller workloads are comprised of similar queries detected by a clustering algorithm that is run over the 6597 queries. In the fifth workload we bundle all the 6597 queries together. Figure 4 displays how the workloads vary in size from 18 to 6597 queries.

Figure 5 and figure 6 show the results of executing the aggregate table recommendation algorithm on these 5 workloads. As demonstrated in these results, the time taken for the algorithm does not have a direct correlation to the input workload size. The algorithm converges to a solution when it reaches a locally optimum solution. When similar queries are clustered together the chances of the locally optimum solution being globally optimum are high. In our experiments when the algorithm is run on all the queries it converges to a globally sub-optimum solution, recommending an aggregate table that benefits fewer queries - and hence has a lower estimated cost saving. The estimated cost savings for each cluster is computed as the sum of the estimated cost savings for each query in that cluster. The estimated cost of each query is derived by computing the IO scans required for each table and then propagating these up the join ladder to get the final estimated cost of the query. The cost savings is the difference in estimated cost when a query runs on base tables versus the aggregated table.

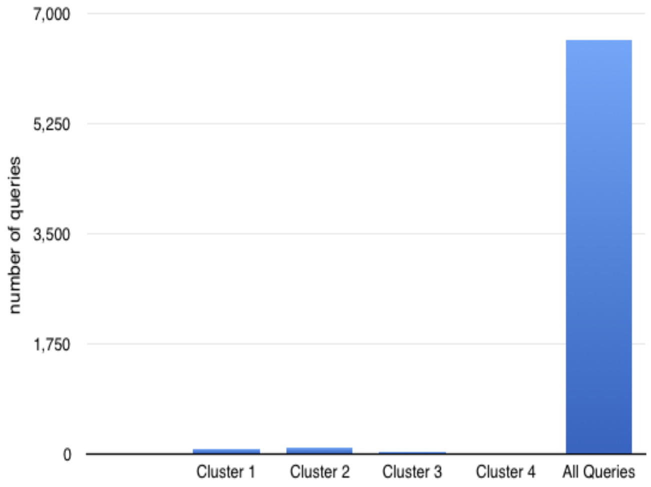


Figure 4: Number of queries per workload.

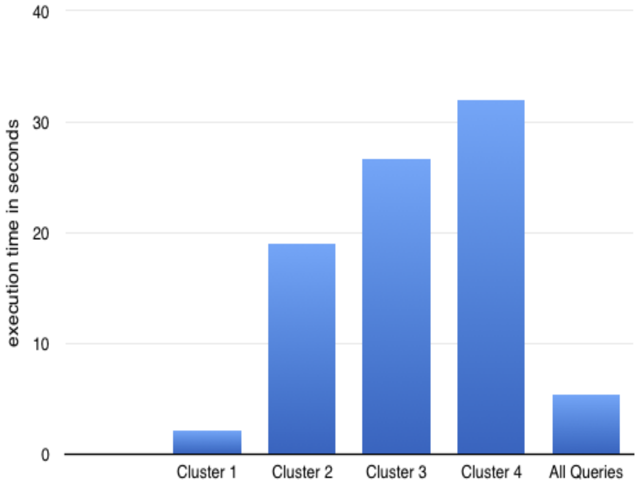


Figure 5: Execution time of aggregate table algorithm.

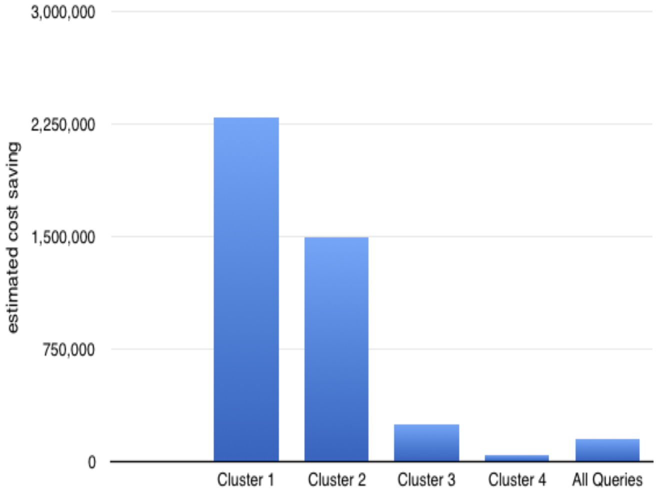


Figure 6: Estimated Cost savings per workload.

Table 3: Merge and Prune

	Execution Time in milli seconds	
Workload Name	With merge and prune	Without merge and prune
Cluster 1	2.092	2.107
Cluster 2	18.919	> 4hrs.
Cluster 3	26.567	> 4hrs.
Cluster 4	31.972	> 4hrs.
Entire Workload	5.279	5.160

4.1.2 Merge and prune

In this set of experiments we evaluate the run time of the algorithm with and without the merge and prune enhancement. We run the algorithm on the same workloads we created for the earlier experiment. We terminated the execution of the algorithm after 4 hours. In the case where the algorithm converges to a solution early on, removal of merge and prune has no effect. But in the other cases the algorithm without merge and prune enhancements takes more than 4 hours to complete and so was terminated. When the algorithm ran to completion without merge and prune, we found no change in the definition of the output aggregate table. The results are tabulated in Table 3.

4.2 Update Consolidation

For update consolidation, we ran our experiments on the TPCH-100 setup. We hand-crafted 2 stored procedures atop TPC-H data inspired from a real world customer workload. The number of consolidations we found in the stored procedure are shown in Table 4. Column 2 shows the number of queries in each stored procedure. Column 3 shows the groups of consolidated queries represented by the index of the query in the stored procedure. We see that sometime there are as many as 14 queries that are consolidated into a single group. We also observed that with templatized code generation, there is a lot of scope for consolidating queries.

For comparison purposes, we take the entire stored procedure and convert the queries inside it to equivalent INSERT/UPDATE queries. Any loops in the stored procedures are expanded to evaluate all updated columns - and consider each one for consolidation. Two-way IF/ELSE conditions are simplified to take all the IF logic in one run, and ELSE logic in the other run. N-way IF/ELSE conditions were ignored.

From our results, we observe that on consolidating 5 queries into a single query, the performance impact is not 5x, for the following reasons:

1. The consolidated CREATE query might have more columns than the individual queries, therefore the amount of data it writes will be larger.

2. If there are few or no common subexpressions, then we re-write the whole table or a significantly higher portion of the table.

But our performance results indicate that even with these caveats, UPDATE consolidation is very efficient compared to individually mapped queries. In all our cases, we found that consolidating even two queries is better than individually executing these queries.

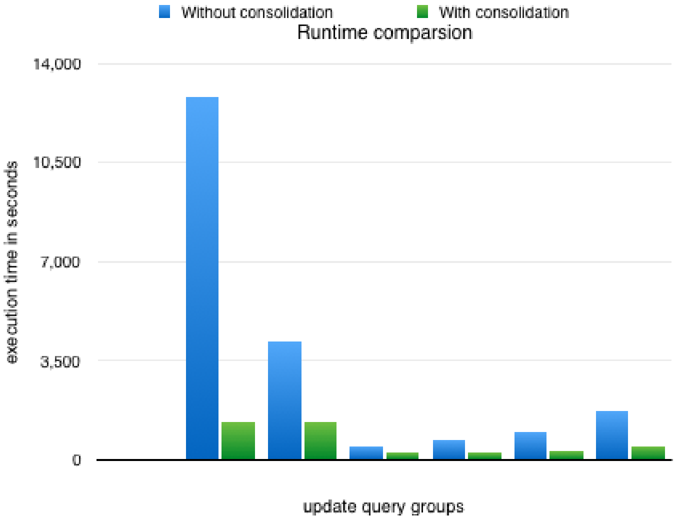


Figure 7: Execution time of consolidated vs non-consolidated queries.

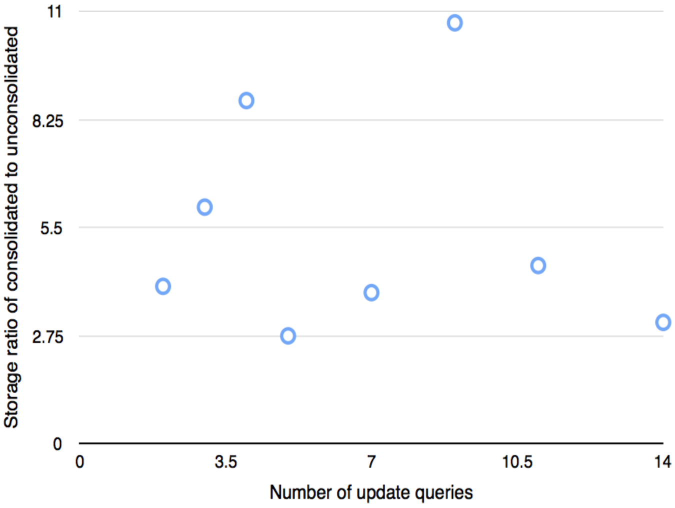


Figure 8: Storage requirements of update queries.

Table 4: Update Consolidation groups

Stored procedure	Number of queries	Consolidation groups
1	38	{6,7,9} ,{10,11} ,{12,14,16,18,20,22,24,26,28} ,{30,32,34,36}
2	219	{113,119,125,131} ,{173,175,177,179,181,183,185,187,189,191,193,195,197,199}

The time taken for detecting UPDATE consolidations is less than a second; hence it was ignored in these measurements. We assume that the database has no triggers, all the tables are independent and updating a table does not incur UPDATEs to tables that are not part of the query. We plot the execution time of non-consolidated queries to consolidated queries in figure 7. The largest group with 14 queries shows a performance improvement of 10x. Even for a group of 2 queries, we see a minimum performance improvement of 80%. The baseline update performance which is spanning few minutes is not an uncommon scenario in SQL-on-Hadoop engines.

The graph in figure 8 shows the storage ratio for consolidated and non-consolidated queries for the size of the consolidation group. If there are multiple groups with the same size, we take the harmonic average of all the groups of the given size. The intermediate storage required for consolidation varies from approximately 2x to as large as 10x when compared to the average storage requirement for individual non-consolidated queries. However in many cases the size of the intermediate table is also significantly less than the original table. In the Hadoop ecosystem, storage is considered a cheap resource and if performance of the UPDATE queries is important, it is certainly worth the trade-off.

These stored procedures are part of a daily workflow that the customer executes, hence the time savings obtained by update consolidation are not only significant but also extremely useful in the big data environment.

5. CONCLUSION & FUTURE WORK

As large scale new and legacy applications are deployed on Hadoop, automated workload-level optimization strategies can greatly help improve the performance of SQL queries. In this paper, we propose creating aggregate tables after first deriving clusters of similar queries from SQL workloads, and demonstrate that in some cases execution time and efficiency improvements of about 1500% can be achieved by using clustered set of queries versus a disparate set of queries as input to the aggregate table creation algorithm. We have also shown empirically that a merge and prune optimization strategy helps the aggregate table creation algorithm converge to a solution, even in cases where it could not converge to a solution. We also showed that 2 to 10X execution time savings can be realized in some cases, when UPDATE queries can be consolidated. Both these optimizations are critical in the Hadoop environment when tables, columns and queries are at very large scales and query response times are of significance.

Advisors [8] [7] [15] [13] [9] rightly emphasize the need for an integrated strategy that evaluates and recommends aggregate table and indexing candidates, together. In the Hadoop ecosystem, partitioning features are the closest logical equivalent to indexes. Currently, if statistical information on a table (such as table volume and column NDVs) is provided, our tool recommends partitioning key candidates for a given table based on the analysis of filter and join patterns most heavily used by queries on the table. We plan to extend this logic to discover partitioning keys for the aggregate tables, thus providing an integrated recommendation strategy.

A further area of focus for the UPDATE consolidation optimization is to explore opportunities to coalesce operations. For example, operations on the temporary table generated in our algorithm can be consolidated to reduce the size of these tables and improve the efficiency of UPDATEs. We are also investigating UPDATE consolidation techniques when UPDATEs are interleaved with control-flow-logic.

Apart from the applicability of our work to adopters of Hive, Impala and Spark SQL who want to optimize their workloads, EDW and BI tools can also use these techniques to improve the efficiency of their workloads.

6. ACKNOWLEDGEMENTS

We would like to thank Prithviraj Pandian for developing the clustering algorithm used in our experiments for Aggregate tables. We would also like to thank our colleagues Justin K, Parna Agarwal, Anupam Singh and Laurel Hale for their insightful review comments.

7. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[2] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[3] S. Chaudhuri, A. K. Gupta, and V. Narasayya. Compressing sql workloads. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 488–499, New York, NY, USA, 2002. ACM.

[4] G. Grahne and J. Zhu. Fast algorithms for frequent itemset mining using fp-trees. *IEEE Trans. on Knowl. and Data Eng.*, 17(10):1347–1362, Oct. 2005.

[5] J. Hyde. Discardable memory and materialized queries, May 2014. Available at http://hortonworks.com/blog/dmmq/.

[6] How-to: Ingest and query "fast data" with impala. Available at http://blog.cloudera.com/blog/2015/11/how-to-ingest-and-query-fast-data-with-impala-without-kudu/.

[7] Database engine tuning advisor overview. Available at https://technet.microsoft.com/en-us/library/ms173494(v=sql.105).aspx.

[8] SQL tuning advisor in oracle SQL developer 3.0. Available at http://www.oracle.com/webfolder/

technetwork/tutorials/obe/db/sqldev/r30/TuningAdvisor/TuningAdvisor.htm.

[9] IBM cognos. Available at http://www.ibm.com/support/knowledgecenter/SSEP7J_10.2.2/com.ibm.swg.ba.cognos.cbi.doc/welcome.html.

[10] JIRA:Add materialized views to HIVE. Available at https://issues.apache.org/jira/browse/HIVE-10459.

[11] Apache kudu. Available at http://kudu.apache.org/.

[12] Kudu impala integration. Available at http://kudu.apache.org/docs/kudu_impala_integration.html.

[13] Microstrategy product documentation. Available at https://microstrategyhelp.atlassian.net/wiki/display/MSTRDOCS/MicroStrategy+Product+Documentation.

[14] Qubole quark. Available at http://qubole-quark.readthedocs.io/en/latest/.

[15] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. Db2 design advisor: Integrated automatic physical database design. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 1087–1097. VLDB Endowment, 2004.