Automated Management of Indexes for Dataflow Processing Engines in IaaS Clouds

Herald Kllapi
Department of Informatics and Telecommunications,
University of Athens, Greece
herald@di.uoa.gr

Ilia Pietri
Department of Informatics and Telecommunications,
University of Athens, Greece
ipietri@di.uoa.gr

Verena Kantere
School of Electrical and Computer Engineering,
National Technical University of Athens, Greece
verena@dblab.ece.ntua.gr

Yannis Ioannidis
ATHENA Research and Innovation Center, Greece
Department of Informatics and Telecommunications,
University of Athens, Greece
yannis@di.uoa.gr

ABSTRACT

Data structures like indexes are typically used to accelerate dataflow execution locating and accessing data more efficiently. The automated management of data structures has been a challenging problem, traditionally constrained by the time and storage required to build and maintain them. As cloud computing is becoming an attractive platform for the execution of dataflows with the usage of compute and storage resources being charged by cloud providers, monetary cost is becoming an equally important factor for the user to consider. In this work, we identify the opportunity of interleaving dataflow and index build operators in the execution schedule to utilize idle slots for the creation of indexes which may be beneficial for future dataflows. In that way, the cost of building indexes can be eliminated without impacting dataflow execution. We propose an online auto-tuning approach to assess the importance of indexes for the workload based on historical data taking into account the trade-off between the dataflow speed-up they offer and the monetary cost needed to maintain them. The results show that the proposed approach can dynamically adapt to the workload and significantly reduce the average execution time and cost spent per dataflow building and maintaining a proper set of indexes.

1 INTRODUCTION

Modern applications face the need to process large amounts of data using complex functions for analysis [40], data mining [32], Extract-Transform-Load processes (ETL) [45], and more. Such rich tasks are typically expressed in high-level languages like Pig Latin [39], optimized and transformed into data processing flows, or simply dataflows, that describe computations (operators) and flow dependencies between them [34],[33],[48].

Dataflows are usually executed on distributed systems to process independent operators in parallel and reduce overall execution time. Among these, clouds have evolved to a popular platform for large-scale data processing, mainly due to the lack of any upfront investment and *elasticity* (the ability to lease resources on demand for as long as needed). Cloud providers offer compute resources in the form of virtual machines (VMs) which are typically charged based on a per quantum pricing scheme (e.g. one hour) such as Amazon EC2 [3], and storage resources which are usually charged per GB per month [5].

Data structures like indexes and materialized views are additionally used to improve the performance of dataflows, encapsulating prior computations to access data more quickly and avoid unnecessarily large data movements during dataflow execution [36]. Building and maintaining indexes may be costly in terms of computation and storage, often exceeding the gain in performance [30]. However, in several cases the costs can be amortized. For example, the time overhead required for the creation of indexes may be reduced by building them in parallel. Also, indexes are usually associated to multiple dataflows and can thus be exploited for the execution of future dataflows. As the existence of indexes may improve application performance, but may also affect the monetary cost incurred [22, 41], it is important to find a good trade-off between these two conflicting objectives. Hence, index tuning (the selection of indexes based on their usefulness) is required to avoid uncontrolled creation and maintenance of data structures. This task may become even more challenging, when the workload is not known a-priori and the set of indexes may change dynamically over time.

We envision a Query-as-a-Service (QaaS) platform to manage the execution of complex dataflow workloads on clouds, like Google's BigQuery[1]. Dataflows, such as exploratory data-intensive queries, are issued sequentially by the user, e.g. a data scientist, to extract knowledge from data. Each dataflow is associated with a set of indexes that can benefit its execution. The service incorporates automated management of suggested indexes by creating and deleting them based on their usefulness on the dataflow workload. These indexes can either be computed automatically or incorporate feedback from administrators to generate useful recommendations [16, 29, 43]. This is an orthogonal problem and the integration of already proposed solutions would easily work with our approach. For example, most index advisors can output a set of indexes that might be useful (e.g., by doing a what-if analysis). This would be the input to our system.

Building a generic model that captures dataflows and indexes is an open research problem, mainly because operators may have arbitrary user code that is often impossible to analyze, and the usefulness of an index may be specific to each dataflow. However, this is beyond the scope of this work. We identify five generic categories of dataflow operators where indexes can be useful:

- *Lookup*. The complexity of finding a particular record from an input table of size n is $O(n)$ when no data structure is used and can be reduced to $O(log\ n)$ using a B-tree index or $O(1)$ using a hash index.

[1] Google Big Query, https://cloud.google.com/products/big-query

- *R*ange select. Selecting records in a particular range from the input can be efficiently performed using a B-Tree index because the leaves of the tree are sorted. The complexity is $O(log\ n) + k$ where k is the number of records in the range.
- *S*orting. The complexity of operators that perform sorting is $O(n \cdot log\ n)$ and can be reduced to $O(n)$ using a B-Tree index.
- *G*rouping. Grouping can be efficiently performed using sorting, as described above.
- *J*oin. Several algorithms, such as nested loops join, hash join, sort-merge join, can be used. Such algorithms are faster when an appropriate index is provided. For example, the complexity of sort-merge join is $O(n + m)$ if the inputs (of size n and m) are sorted.

In this work, we propose an online auto-tuning approach to assess the usefulness of indexes for the execution of dataflows taking into account the trade-off between the dataflow speed-up they offer and the monetary cost needed to maintain them (the storage cost in the cloud). We identify the opportunity to build indexes and eliminate their cost using slots of *idle time* on compute resources. These may be created due to data dependency constraints between the execution of dataflow operators but also the provider's quantized pricing policy. Building entire indexes sequentially using idle compute resources may not be feasible due to the large data volume [41]. Hence, indexes on partitions of tables or files are built independently. In this way, indexes can be built in parallel and, most importantly, may fit inside idle slots. The approach proposed in this work is generic and can be used in several large-scale data processing platforms, like Hadoop [7], Hive [46], or Pig [39]. Several systems like [26, 35, 46] have been developed to provide highly scalable distributed architectures for data processing on the Cloud; however, the monetary cost and the quantized pricing of resources need to be considered [23]. To the best of our knowledge, there is no index management solution that takes into account the monetary cost of using cloud resources, while related work on execution time and cost optimization of dataflows does not consider building and maintaining indexes.

The main contributions of this work are the following:

- We identify the opportunity to use idle slots on compute resources created when executing data-intensive flows due to data dependency constraints between operators but also the quantum-based pricing policy of compute resources.
- We propose an online auto-tuning approach to assess the importance of indexes based on the trade-offs between the dataflow execution speed-up they offer and the monetary cost needed to maintain them.
- We develop two index interleaving algorithms, namely linear program based interleaving and online interleaving algorithms, to utilize idle slots in the dataflow execution schedule and build indexes in parallel without increasing the monetary cost and the time required for the execution of each dataflow.
- We provide an experimental evaluation to show the effectiveness of the proposed approach to accelerate dataflow execution and eliminate the related monetary costs.

The rest of the paper is organized as follows. Related work is discussed in Section 2. The problem description follows in Section 3, while the optimization problem is defined in Section 4. The online auto-tuning approach and interleaving algorithms proposed are described in Section 5. The experimental evaluation and its results follow in Section 6, while Section 7 concludes the paper.

2 RELATED WORK

A considerable body of work focuses on VM consolidation to exploit underutilized resources for the execution of multiple workloads [14, 51]. However, consolidating different workloads may greatly affect application performance due to interference, as consolidated VMs may compete for resources [53]. In contrast, the idea of this work is to interleave dataflow and index build operators in the execution schedule to accelerate dataflow execution while eliminating the cost of building indexes.

Offline algorithms for index tuning on centralized systems like [10, 16] do not consider a dynamic environment where the service is unaware of the dataflows and a priori predictions of how long to keep and when to delete indexes cannot be made. Our approach is closer to online algorithms like [9, 38, 52]. However, we target a distributed and elastic environment where VMs are allocated dynamically and compute resources are prepaid for the whole time quanta. Also, what-if optimizations that improve index tuning [16] are complementary to our work and can be used to accelerate the computation of index usefulness. Approaches that incorporate feedback from administrators to improve index recommendations [29, 43] are also orthogonal to our work, as user feedback can be beneficial for the computation of index usefulness. The problem of index interactions has also been studied [42, 44]. Such efforts could be leveraged in our work to delete indexes that become obsolete when index interactions in the dataflow workload are identified.

Online algorithms for distributed environments like [13, 20, 41, 47] focus on replicated databases, investigating which sets of indexes to build on each replica and how to route queries properly to take advantage of them. Such approaches can be used in combination with our proposed approach since multiple replicas for each partition are typically created in distributed environments to increase efficiency and fault tolerance [24]. Indexing mechanisms on clouds like [11, 15, 36] mainly focus on the optimization of application performance and ignore the monetary cost of using the resources. The monetary cost of data structures has been considered in multi-user environments [30, 49] to distribute the creation and maintenance costs of data structures among multiple users. However, our work focuses on single-user environments where resources allocated to the user are dedicated and data structures built are not shared among multiple users. This way, each user is independent and the provider's pricing policy for compute and storage resources like Amazon Elastic MapReduce [4] can be directly used, without considering complex cost sharing policies that users may or may not agree with. Finally, the work in [21] considers the problem of data structure reuse by future queries, materializing and storing the output of operators of MapReduce jobs. To the best of our knowledge, there is no index management approach for single-user environments that takes into account the monetary cost of using cloud resources.

3 PROBLEM DESCRIPTION

Figure 1 shows the architectural framework envisioned in this work. The typical users of the QaaS service are data scientists that issue exploratory query tasks to extract knowledge from data, such as data intensive transformations that perform processing and aggregations on data read from tables or files. The data can be partitioned for flexibility, performance, and fault tolerance and indexes on each partition of tables or files can be built. Each user interacts with the service by issuing dataflows sequentially,

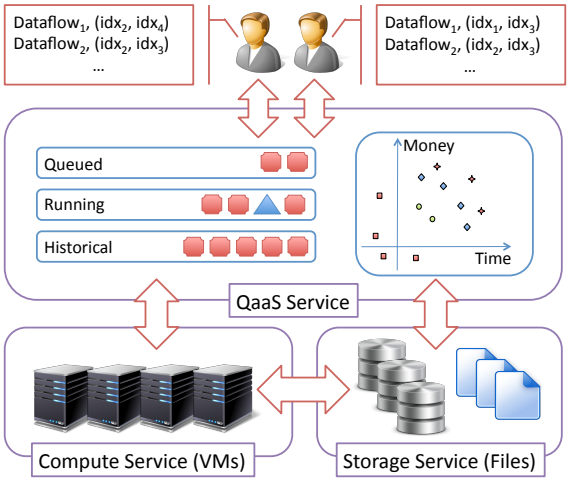
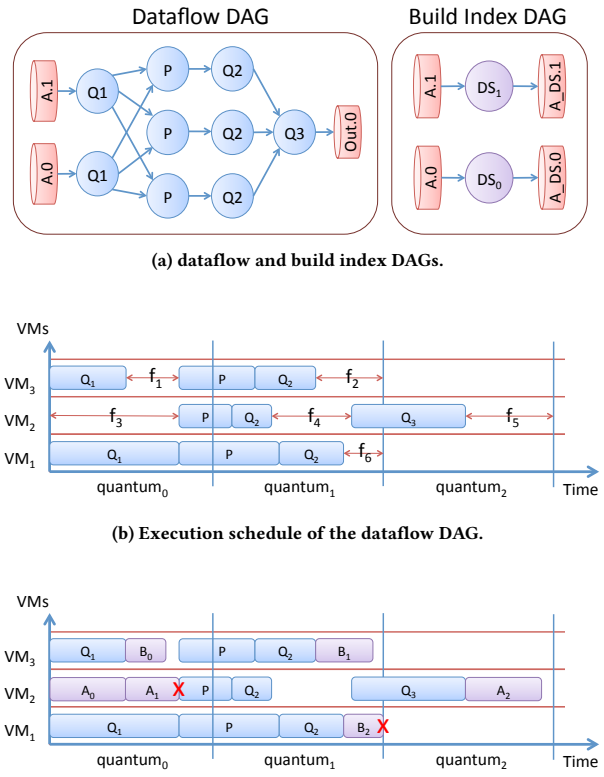


Figure 1: The setting of the QaaS service.

usually observing the results obtained from the execution of a single dataflow before submitting the next one. The service executes the dataflows on top of clouds according to selected execution schedules with desired time-money trade-offs using compute and storage services offered by cloud providers. The execution of the dataflow is interleaved with the execution of build index operators and the indexes created are stored in the cloud storage. Each dataflow submitted for execution has access to currently available indexes and each operator can make use of those associated to partitions it accesses.

Motivation. A dataflow example is shown in Figure 2a. As can be seen in the graph of the DAG (left part of the figure), the dataflow uses two partitions, $A.0$ and $A.1$, of an input table A and performs processing ($Q1$, $Q2$), partitioning (P), and aggregations ($Q3$). The dataflow is associated to a set of indexes ($A\_DS.0$ and $A\_DS.1$) built for the table partitions ($A.0$ and $A.1$, respectively) as shown in the right part of the figure. There are two indexes to be built: A built in three parts $A0$, $A1$ and $A2$ and B built in parts $B0$, $B1$ and $B2$. Parts can be created in parallel using different VMs. Figure 2b shows an execution schedule of the dataflow operators when using 3 VMs ($VM1, VM2, VM3$). Arrows show the idle slots created due to data dependency constraints and the quantized pricing policy ($f1 - f6$). For example, slot $f4$ on $VM2$ remains idle as $Q3$ cannot be executed until all $Q2$ operators have finished. Such idle slots can be used for the building of indexes without incurring any additional cost, as shown in Figure 2c. Different indexes can be built in parallel such as the case of A_1 and B_0. The execution of the index build operator A_1 at $VM2$ is stopped as it is not completed before the execution of the dataflow operator P starts so that the execution of the dataflow is not delayed. Similarly, B_2 is stopped before the time the leased quantum expires to avoid unnecessary costs for building indexes.

Application Model. A dataflow d is modelled as $d(expr, R, N, t)$, where $expr$ is its definition expressed in an appropriate language, R is the set of input tables, N is the set of indexes that can accelerate its execution, and t is the time point that the dataflow is issued to the service. The dataflow is modelled as a DAG where the nodes correspond to operators and the edges to data dependencies (flows) between them. An *operator* is modelled as $op(cpu, memory, disk, time)$, where cpu is the CPU utilization,



(a) dataflow and build index DAGs.

(b) Execution schedule of the dataflow DAG.

(c) Interleaving of dataflow and build index operators.

Figure 2: Execution of an example dataflow and indexes

mem is the maximum memory needed for its normal operation, *disk* is the disk resources, and *time* is its execution time. A flow between two operators is labelled with the size of the data transferred between them. The estimations of operators can be computed analytically or can be collected by the system at runtime [37]. Since we target large datasets, the statistics (e.g., histograms) do not change radically over time (a 10GB update on 1TB dataset is not large enough to change the statistics). The dataflow processing rate is much higher than the rate at which the data is updated. This is the typical case in many settings: updates are done every few days and the datasets are processed much more frequently [27]. Also, operators come from a set that does not change frequently, which is typical for exploratory data analysis [36].

Cloud Model. Compute resources are offered in the form of VMs (or containers) with a fixed capacity of resources, CPU, memory, disk, and network, respectively. Each VM is charged at a fixed price (M_c) per time quantum (Q) and can be dynamically allocated and deallocated based on the workload needs. In this work, homogeneous VMs are assumed. This is typical for many installations; most VMs are of the same size and only few VMs which run critical services are significantly larger (like namenodes of Hadoop [7]). An idle VM (a VM that is not used) is deleted when its currently leased time quantum expires, since the resources are prepaid for whole time quanta [3]. Each VM has a local disk that can be used to store temporary results or data. After deleting a particular VM, the files stored in its local disk cannot be recovered.

A storage service is used to store data persistently; VMs retrieve data from the storage service and cache it to their local disks and transfer data to the storage service after the execution of an operator finishes. This scheme is flexible as compute from storage resources are decoupled. Typically, cloud providers charge a fixed amount of money per GB per month (MC). In the model used, the cost of storing data, M_{st}, is measured in GB per time quantum (Q). As a year has approximately 365.25 days and, assuming that Q is measured in minutes, we compute M_{st} as $(MC \cdot 12 \cdot Q)/(365.25 \cdot 24 \cdot 60)$.

Data Model. Tables can be partitioned and stored to the storage service of the cloud. As mentioned, allocated VMs can cache table partitions to their local disk to avoid network overheads when possible. Data updates are performed in batches periodically (every day or week). Each update creates a new version of the table partitions changed [2], invalidating old versions and indexes built on them. A table t in the database is modelled by its schema (i.e., the names and types of its columns), its ordered set of partitions, and its statistics: $t(schema, P, S)$. A partition $p \in P$ is modelled by its id, its number of records n and a particular path in the storage service where the partition is located: $p(id, n, path)$. The statistics contain the average size of the fields of each column.

An index idx built on table t is modelled as $idx(t, C, T)$, where C is the ordered set of columns based on which the index is built and T is the ordered set with the respective creation time points of its partitions. Each index consists of several index partitions built on different table partitions. Index partitions can be built on different time periods. The *index size* is computed by adding the sizes of its partitions. The size of a partition can be computed based on the type of the index (e.g. Hash, B^+Tree). We assume without loss of generality that B^+Trees are used. The size of partition p of index idx is computed as follows:

$$size(idx, p) = (1 - k^{\log_k(p.n)}) \cdot RecSize/(1 - k),$$

where $RecSize$ is the average size of the record in the index, computed from column statistics, and k is the width of the tree computed from the block size on the disk and the record size $RecSize$. Assuming that the tree is balanced, its size is computed using geometric series as follows: the total number of records including the non-leaf blocks is $\sum_{i=0}^{m} k^i = (1-k^m)/(1-k)$, where m is the height of the tree computed as $m = \log_k N$. Parameter N represents the number of records in the partition. The time to build an index idx, $t_i(idx)$, is computed by adding the time to build all the index partitions of the corresponding table. The time to build the index on a partition p is computed as:

$$t_{ip}(idx, p) = t_{io}(idx, p) + C(idx) \cdot p.n \cdot \log_k(p.n)/T_Q,$$

where $C(idx)$ is a constant calculated using the columns in the index. The time to read and write the partition $t_{io}(p)$ is computed as:

$$t_{io}(idx, p) = (p.n \cdot RecSize + size(idx, p))/cont.net,$$

where $cont$ is the container to which the build index operator is assigned for execution. The building of indexes can be expressed as a DAG with operators that take as input one partition and build the partial index on that partition. Operators are independent to each other (there are no edges between the operators in the DAG) and as a result there is a large degree of parallelism. Hence, indexes can be built incrementally (not all index partitions need to be built in order to use the index) and in parallel (two or more index partitions can be built simultaneously). The storage cost of index idx for a time period W (given in time quanta) is computed by adding the cost $stp(idx, p, W)$ of storing each index partition p for that period, where

$$stp(idx, p, W) = W \cdot size(idx, idx.t.P[p]) \cdot M_{st}.$$

Our approach can work with different pricing models. A pricing model is plugged to the scheduler by using the appropriate pricing formulas for the cost of a VM (M_C) and the cost of storage (M_{st}). Also, although we consider a homogeneous environment with a single VM type, the scheduler can consider slots at different VM types.

Dataflow and Index Management. The dataflows are issued sequentially to the service. Historical dataflows (dataflows that have already been executed) are stored in a list called H_d. An execution schedule S_d of a dataflow graph d is a set of assignments of its operators to containers. An execution schedule is computed taking into account the network communication cost using the model in [33]. The execution time of a dataflow in schedule S_d, $t_d(S_d)$, is defined as the time period from the time the first operator starts executing until the time the execution of the last operator finishes. The monetary cost $m_d(S_d)$ is computed taking into account the sum of the total time quanta of the VMs leased. The monetary cost and execution time are measured in quanta in order to have the same unit. An idle slot $f(id, q, c, S_d)$ in schedule S_d is a continuous time period inside the leased time quantum q of the container, c, that has no operators running. The fragmentation of the schedule is the set of all the idle slots in the leased containers and shows the time the compute resources are not used during the execution schedule, but they are charged by the cloud provider.

Idle slots can be used for the building of indexes. We denote as I the evolving ordered set of indexes built and maintained by the service. The set of indexes available at time t is denoted as $I(t)$ and the set of all indexes created during the operation of the service (independently of whether they are deleted or not) is denoted as I. Potential indexes are indexes that are associated with one or more dataflows, but they are not beneficial to build. Indexes built on table partitions that are updated are deleted and marked as *not built* to support index updates.

4 OPTIMIZATION PROBLEM

This work considers the problem of interleaving indexes with the execution of dataflows so that dataflow execution, in terms of execution time and monetary cost, is not affected. The aim is to determine the set of beneficial indexes required to build and maintain over time to achieve good trade-offs between the dataflow speedup and the monetary cost required taking into account the storage cost needed to maintain the indexes.

The optimization problem is formulated as a weighted single objective problem using a linear function to express the different tradeoffs between the time and money objectives:

$$\max_I \Big[\sum_i M_c \cdot (\alpha \cdot \delta t_d(d_i) + (1-\alpha) \cdot \delta m_d(d_i)) - \sum_j st(I[j]) \Big], \quad (1)$$

where d is the dataflow, $st(I[j])$ is the storage cost of index $I(j)$, $\delta t_d(d_i)$ is the difference (given in quanta) in dataflow execution time without and with the use of indexes and $\delta m_d(d_i)$ is the difference (quanta) in the monetary cost required without and with the use of indexes. Parameter α essentially expresses how much money a time quantum is valued, taking values between 0 and 1 that correspond to scenarios where the optimization

Table 1: Notation used.

Parameter	Meaning
T_Q	Quantum size
M_c	VM price (per quantum)
M_{st}	Storage cost (per GB per quantum)
$I(t)$	The set of indexes available at time t
a	Parameter for time-cost trade-off
$st(idx, W)$	Storage cost of index idx within a time window W
$gt(idx, t)$	Gain in time for index idx at time t
$gm(idx, t)$	Gain in money for index idx at time t
$dc(t)$	Gain fading function
$t_i(idx)$	Time for building index idx
$m_i(idx)$	Monetary cost for building index idx

problem becomes one-dimensional. Small values of α indicate that monetary cost (or money) is more important, while large values of α indicate scenarios where time is more important. The difference in time $\delta t_d(d_i)$ is multiplied with the container price per quantum (M_c) so that the time and money objectives have the same units.

In a dynamic environment where arbitrary dataflows are issued at arbitrary time points using different sets of potential indexes, it is hard to find the optimal sequence of index sets ($I(t)$), i.e., determine when to build and delete indexes using Equation 1. We formulate the optimization problem to a more suitable form (Equation 2) for the computation in an online fashion:

$$I(t) = arg\max_{I} \left[\sum_{idx \in I} (\alpha \cdot M_c \cdot gt(idx, t) + (1-\alpha) \cdot gm(idx, t)) \right], \quad (2)$$

where functions $gm(idx, t)$ and $gt(idx, t)$ (described in Equations 4 and 5) compute the gain in money and time, respectively, when using a particular index idx at time point t and within a time window of predefined size W (e.g., two quanta). Table 1 summarizes the notation used to define the optimization problem. Equation 2 can be approximated in an online fashion by computing the gain of each index individually, building and maintaining only those that contribute in a positive way to the summation at any given point in time. More specifically, an index idx is said to be beneficial at time point t if its gain (the weighted summation of time and money gain of Equation 3) is positive.

$$g(idx, t) = \alpha \cdot M_c \cdot gt(idx, t) + (1 - \alpha) \cdot gm(idx, t) \quad (3)$$

Indexes are built as soon as they become beneficial and are deleted as soon as they become non beneficial.

The gain in money $gm(idx, t)$ of index idx is computed by adding the gain in money $gm_d(idx, d_i)$ of the index on each related dataflow d_i (the dataflows that use it and are evaluated inside time window $[t-W, t]$ and the currently running dataflow) and the monetary cost required to build and store the index for time period W, as described in Equation 4:

$$gm(idx, t) = \sum_i \left(\delta(d_i, t) \cdot dc(\delta T_{d_i}) \cdot M_c \cdot gm_d(idx, d_i) \right) - (M_c \cdot m_i(idx) + st(idx, W)) \quad (4)$$

where $\delta(d_i, t)$ is 1 if the dataflow f has been executed during time period $[t - W, t]$ or else 0, δT_{d_i} is the number of quanta passed since the dataflow d_i was executed (0 for the ones that are currently running or queued) and $dc(t)$ is a function that reduces with time in order to fade the gain of the historical dataflows. An exponential function is used to fade the gain: $dc(t) = e^{-t/D}$, where parameter D controls the degree the historical dataflows affect it. A small value of D means that $dc(t)$ approaches quickly to 0 and, as a consequence, $gm(idx)$ becomes negative. We assume

Table 2: Dataflows Issued using Indexes A and B.

Dataflow	Time Gain	Money Gain
$d_1(-, -, \{B\}, 10)$	$gt_d(B, d_1) = 1.0$	$gm_d(B, d_1) = 3.0$
$d_2(-, -, \{B\}, 30)$	$gt_d(B, d_1) = 2.0$	$gm_d(B, d_1) = 5.0$
$d_3(-, -, \{A, B\}, 50)$	$gt_d(A, d_1) = 2.0$ $gt_d(B, d_1) = 3.0$	$gm_d(A, d_1) = 8.0$ $gm_d(B, d_1) = 8.0$
$d_4(-, -, \{A\}, 100)$	$gt_d(A, d_1) = 3.0$	$gm_d(B, d_1) = 5.0$

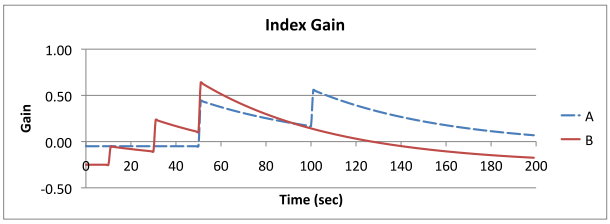


Figure 3: Gain over time of two indexes A and B.

that D is the same for all indexes. However, different values of the controller D can be used for each individual index. Automatic learning of the controller for each index based on predictions is a direction for future work. Also, $m_i(idx)$ is the monetary cost required to build the index, $st(idx, W)$ the storage cost required to maintain it for a time window W and $gm_d(idx, d_i)$ is the gain in money of dataflow d_i when using index idx which is computed based on the time gain of the index on d_i. The gain in money $gm_d(idx, d_i)$ also includes the monetary cost spent to read the index from the storage service, which is equivalent to the time to read the index, as both of them are measured in quanta. If dataflow d_i does not use index idx, then $gm_d(idx, d_i) = 0$.

Similarly, the time gain $gt(idx, t)$ of index idx at time point t is computed taking into account the gains of index idx on the dataflows executed within the time window W, subtracting the time needed to build it as follows:

$$gt(idx, t) = \sum_i \left(\delta(d_i, t) \cdot dc(\delta T_{d_i}) \cdot gt_d(idx, d_i) \right) - t_i(idx) \quad (5)$$

where $gt_d(idx, d_i)$ is the gain in time of dataflow d_i when using index idx.

An example to illustrate the proposed approach is presented. Assume the dataflows shown in Table 2 are issued to the service at the time points specified. The dataflows use two indexes, A (of size 100 MB) and B (of size 500 MB). The time and money gain of the indexes for each dataflow is included in the table. Figure 3 shows the gain of each index computed over time for the case of $\alpha = 0.5$ and $D = 60$. It can be seen that the gain of both indexes, A and B, is negative in the beginning due to their storage cost. As dataflows specify them as useful, the gain becomes positive at some point (the indexes become beneficial) and then decreases over time because of parameter D (that impacts index usefulness). For example, index B becomes beneficial at time point 30 and will be deleted at time point 125 where it stops being useful.

5 AUTO-TUNING APPROACH

In this section, we propose an auto-tuning approach to select and build an optimal set of indexes over time. Statistics from historical (issued) dataflows and their specified indexes are continuously collected and used to make decisions about which indexes to build or delete at each time point. Dataflows to be executed can only use indexes that are currently available, while new indexes to be built are scheduled with the currently issued dataflow. Indexes are built using idle slots in the execution schedule of the issued dataflow so that the dataflow execution is not affected. However, beneficial

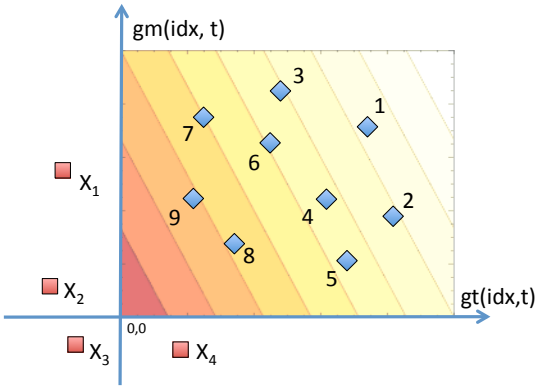


Figure 4: Index ordering based on α at time point t.

indexes to be built may not fit in the currently available slots and selecting which beneficial indexes to build is required. Since the goal is to maximize the optimization objective of Equation 2, indexes are ranked based on their usefulness and the best subset to be built is selected.

5.1 Index Ranking

Equation 3 is used to compute the usefulness or gain of indexes at each time point, as described earlier in Section 4. Non beneficial indexes are not built or are deleted if they are already available. Note that an index that is not used for a long time may become non-beneficial because of the increased storage cost. We only consider beneficial indexes with $gm(idx, t)$ and $gt(idx, t)$ (Equations 4 and 5) larger than 0 and among them higher values of gain (Equation 3) are preferred. Essentially, indexes can be depicted in a bi-dimensional space based on their time and money gain as shown in Figure 4. Indexes at the lighter areas are prioritized. For example, point 1 has the highest priority for a=0.7, while indexes X_1, X_2, X_3, and X_4 are not beneficial.

5.2 Online Index Tuning

The online index tuning approach proposed is shown in Algorithm 1. The algorithm schedules the issued dataflow along with the subset of potential indexes that maximize the total gain. Beneficial indexes are assigned to idle compute resources in the dataflow execution schedule without violating the constraints (i.e, the time and the monetary cost of the dataflow are not affected). Indexes that are not beneficial or cannot fit to the schedule are deleted. Note that partitions of a particular index can be built in the context of several dataflows if there is not enough idle time to build it entirely in the context of one dataflow. The algorithm is triggered every time a new dataflow is issued, the execution of a dataflow finishes or periodically at fixed time intervals to delete indexes that become non beneficial when there is not any new dataflow to be issued. In more detail, the procedure triggered is the following. The gains in time and cost for each index are computed and beneficial indexes are ranked (lines 2-9 of Algorithm 1) as described in Section 5.1. Then, the algorithm calls the index interleaving procedure to compute the skyline of execution schedules of the dataflow df interleaved with build index operators and selects from the skyline the schedule to be executed (lines 10-11). Different methodologies can be used to choose the schedule to be executed. In this work, the fastest schedule is chosen. In lines 13-19 the algorithm identifies index partitions that are not beneficial and need to be deleted.

Algorithm 1 Online Index Tuning

Input:
H_d: The historical dataflows.
A_i, B_i, P_i: The index lists.
df: The next dataflow to schedule.
Return:
S_{df}: The schedule of the dataflow.
S_{BI}: The schedule of the build indexes.
DI: The indexes that should be deleted.

1: $GAINS \leftarrow \emptyset$
2: **for** $i \in P_i$ **do**
3: $gt \leftarrow gt(i, H_d \cup df)$
4: $gm \leftarrow gm(i, H_d \cup df)$ ▷ Compute the index gains
5: **if** $gt > 0$ and $gm > 0$ **then**
6: $GAINS \leftarrow GAINS \cup \{i\}$
7: **end if**
8: **end for**
9: $RANK \leftarrow rank2Dspace(GAINS)$ ▷ rank the indexes
10: $skyline \leftarrow schedule(df, A_i, RANK)$ ▷ Scheduling of both the dataflow and indexes
11: $S_{df}, S_{BI} \leftarrow select(skyline)$ ▷ Select the schedule from skyline
12: $DI \leftarrow \emptyset$
13: **for** $i \in A_d$ **do**
14: $gt \leftarrow gt(i, H_d \cup df)$
15: $gm \leftarrow gm(i, H_d \cup df)$
16: **if** $gt \leq 0$ and $gm \leq 0$ **then**
17: $DI \leftarrow DI \cup \{i\}$ ▷ Indexes to be deleted
18: **end if**
19: **end for**
20: return (S_{df}, S_{BI}, DI)

5.3 Index interleaving approaches

In this section, we propose two different approaches to schedule dataflows interleaved with build index operators without using additional monetary cost, namely the Linear program based interleaving algorithm (LP) and the online interleaving algorithm. The LP interleaving algorithm initially schedules the currently issued dataflow and finds the idle slots in the compute resources. Then, it uses a linear programming algorithm to determine the subset of potential index partitions and tries to assign them on the idle slots based on their ranking (gain). The online interleaving algorithm schedules the current dataflow and the index build operators together labeling the index build operators as optional operators to be scheduled.

5.3.1 Linear program based interleaving algorithm. The LP interleaving algorithm shown in Algorithm 2 schedules indexes after the dataflows. More specifically, the algorithm initially updates the operator runtimes based on the available index partitions. Estimations of runtimes can be provided based on existing models [50]. The algorithm calls the scheduler described in Algorithm 4 to compute the skyline of the execution schedules (line 6). For each schedule in the skyline, the algorithm finds the set of idle slots and sorts them in decreasing order based on their size (lines 8-10). For each slot, a linear program (line 12) is solved to determine the subset of potential indexes that maximize the total gain. The build index operators in each idle slot are sorted by gain so that the building of less useful indexes is stopped when the time quantum ends or the next assigned operator is scheduled (as shown in Figure 2c) before the build index operator finishes due to runtime estimation errors. The build index operators whose execution has been stopped are queued and scheduled with the next dataflow issued. Overall, the algorithm does not violate the constraints (i.e. index interleaving does not affect dataflow execution in terms of time and money) as indexes are built on slots that are not used for the execution of dataflow operators, but they are charged.

Algorithm 2 Linear program based interleaving algorithm

Input:
df: The current dataflow from the input.
A_i: The available indexes.
I: The ranked list of indexes.
Return:
$skyline$: The skyline of solutions.

1: **for** $op \in df$ **do**
2: **if** op uses indexes in A_i **then**
3: update(op, A_i) ▷ op. runtimes based on available indexes
4: **end if**
5: **end for**
6: skyline ← Skyline(df) ▷ generate skyline of execution schedules
7: **for** s in skyline **do**
8: idle_time ← FindIdleTime(s)
9: ordered_idle_time ← OrderBySize(idle_time)
10: indexes ← ∪($I.P$)
11: **for** i in ordered_idle_time **do**
12: maxset ← SolveLinearProgram(i, indexes)
▷index set to be built based on linear program
13: **for** m in maxset **do**
14: schedule(m, i) ▷ assign indexes to idle slots
15: **end for**
16: indexes ← indexes - maxset
17: **end for**
18: add $indexes$ to s
19: **end for**
20: return skyline ▷ schedules of dataflow ops and index build ops

Algorithm 3 Linear Program Algorithm

Input:
f: The size of the idle time segment.
p_i: The sizes of all the build index partition operators.
g_i: The gain of all the build index partition operators.
Return:
The subset of the build index operators that maximize Equation 2

1: $max\left[\sum_i(w_i * g_i)\right]$
w.r.t
2: $\sum_i(w_i * p_i) \leq f$
3: $0 \leq w_i \leq 1, \forall i$
4: $integer(w_i), \forall i$
5: return $(w_1, w_2, \ldots w_n)$

Linear program approximation algorithm. The problem of assigning build index operators into idle time slots on compute resources is a variation of the Knapsack [31] problem, which is NP-hard. The Linear program approximation algorithm shown in Algorithm 3 is an approximation algorithm to solve a 0/1 knapsack problem for each idle time slot. The algorithm solves the relaxed problem setting the weights of the build index operators between the values 0 and 1 and calls a branch and bound algorithm to find integer values.

Skyline dataflow scheduler. Different execution schedules that vary in the achieved execution time and monetary cost can be created by assigning the dataflow operators to potential slots of the available VMs. Between them, non-dominated solutions (solutions that outperform others in terms of execution time and monetary cost) may be preferred. The set of non-dominated solutions achieved comprises the obtained *skyline* of execution schedules. The algorithm in [12] is used to develop the skyline of execution schedules for each dataflow. An operator is candidate for assignment when all of its predecessors are assigned, starting from operators without data dependencies (entry nodes in the dataflow graph). At each iteration, the algorithm (Algorithm 4) assigns the next available operator to the partial solutions of the current skyline taking into account the communication costs and data dependency constraints between the operators. After the assignment of the new operator to all the possible slots, the new skyline is computed. Between schedules with the same execution time and monetary cost, the schedule with the most sequential idle compute time is selected, since the aim of our work is to use idle slots where index build operators may fit. The procedure described is repeated for the next available operator. The algorithm terminates when all operators are assigned and the final skyline is generated. Note that the impact of data transfers on the execution of data-intensive dataflows may be significant and overhead may be introduced [18]. Thus, each dataflow is scheduled offline to generate more efficient schedules where the overhead from data transfers is considered.

Algorithm 4 Skyline Dataflow Scheduler

Input: df: The dataflow DAG.
C: The maximum number of containers to use.
Output: $skyline$: The solutions in the skyline.

1: $skyline \leftarrow \oslash$
2: $ready \leftarrow$ {operators in df that have no dependencies}
3: $firstOperator \leftarrow ready.peek()$
4: $firstSchedule \leftarrow \{assign(firstOperator, 1, -, -)\}$
5: $skyline \leftarrow \{firstSchedule\}$
6: **while** $ready \neq \oslash$ **do**
7: $next \leftarrow ready.peek()$
8: $S \leftarrow \oslash$
9: **for** all schedules s in $skyline$ **do**
10: **for** all containers c (c ≤ C) **do**
11: $S \leftarrow S \cup \{s + assign(next, c, -, -)\}$
12: **end for**
13: **end for**
14: $skyline \leftarrow$ skyline of S ▷ new skyline of schedules
15: $ready \leftarrow ready - \{next\} \cup$ {operators in df that dependency constraints no longer exist}
16: **end while**
17: **return** $skyline$

5.3.2 Online interleaving algorithm. The online interleaving algorithm is a modification of the scheduler in [12] to use optional operators and schedule index build operators along dataflows. To do so, operators are separated to optional and non optional using a boolean variable; the variable is set to *true* (optional operators for execution) for each index build operator while the variable is $false$ for all dataflow operators. Algorithm 4 is modified so that the schedules in each iteration may vary in the number of assigned operators. The ready operators list (line 2 of Algorithm 4) includes optional index build operators which are candidate for scheduling. If the operator *next* in line 7 is optional, the previous skyline (*skyline*) is kept and unioned with the set of schedules S (line 11) before computing the new skyline in line 14. As a result, the newly generated skyline may consist of schedules with different numbers of operators. Between schedules with the same execution time and money, schedules with a larger number of operators are preferred. Also, the schedules kept in the new skyline do not violate the constraints of the optimization problem, as solutions that belong to the initial skyline and have lower execution time or monetary cost will dominate solutions in the unioned set where the assignment of optional operators have affected dataflow execution. Hence, only schedules where the assignment of optional operators does not affect the dataflow execution time and monetary cost will be kept in the newly computed skyline.

6 EXPERIMENTAL EVALUATION

In this section, the proposed approach is evaluated based on simulation. The skyline dataflow scheduler described in Section 5.3.1 (*offline*) is evaluated using an online load balance scheduler (*online*) typically deployed in elastic clouds as baseline. The *online* algorithm examines the dataflow graph in an online greedy

Table 3: Experiment Parameters

Parameter	Values
Quantum size	60 seconds
Quantum cost	\$0.1
Storage Cost	$\$10^{-4}$ per MB per Quantum
Max Containers	100
Dataflow	Montage, Ligo, Cybershake
Operators / Dataflow	100
α	0.5
Index gain fading D	1 quantum
Poisson Generator λ	1 quantum
Total Time	720 quanta

fashion scheduling the operators to the available containers so that load balance is achieved. Finally, the two index interleaving algorithms, the *LP interleaving* algorithm and the *online interleaving* algorithm, are evaluated and compared using two baseline index management approaches: a naive approach that does not create indexes at all (*no indexes*) and an approach that randomly selects indexes from the potential set and randomly assigns them to containers to be built (*random*).

6.1 Experimental Setup

Table 3 summarizes the parameters used in the experiments. Homogeneous containers with similar capacity in resources (CPU, memory, disk, and network) are assumed. Each container has one CPU and one disk. The CPU and memory needs of each operator is specified as a percentage of container's CPU and memory respectively. A disk size of 100 GB and a speed of 250 MB/sec (typical SSD) are assumed. Allocated containers cache table partitions and indexes read from the storage service. A time quantum Q of 60 seconds is assumed. Pricing is based on *Amazon*'s billing policy [6]. The price M_c charged for the provisioning of each container per time quantum is set equal to \$0.1 and the storage cost M_{st} is set equal to $\$10^{-4}$ per MB per quantum. The storage of the cloud is computed by counting the number of bytes transferred and charging appropriately over time.

In the simulator used, user queries are sent to the scheduler, which adds them to a queue. Each query is transformed into an execution graph of operators with data dependencies. Given the execution graph, the scheduler selects a subset of containers and schedules the execution of the graph operators on these containers, respecting the graph dependencies. The set of active containers can be dynamically varied based on the demand. Each operator has a priority specified and each container has a queue with operators that are executed as soon as the memory needed is sufficient. Dataflow operators have priority 1 and build index operators have priority -1. Operators with negative priority are stopped when operators with positive priority arrive to the container or its current time quantum expires. A network bandwidth of 1 Gbps is assumed. The execution of an operator is delayed until its input data are transferred. Also, if an index is available and beneficial, the container reads the index in addition to the input of the operator, depending on the speedup it offers.

In the simulator used, each container has a local disk to cache input files from the storage service. If the data required as input from the operator are already in the cache, data transfer is considered to be 0. If the container cache gets full, LRU policy is used to create empty space. Containers that do not have any dataflow operators scheduled on them are deleted at the end of the leased quantum.

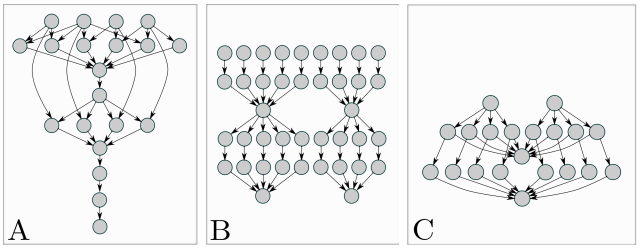


Figure 5: The dataflow graphs Montage(A), Ligo(B), and Cybershake(C).

Table 4: Basic statistics of the scientific dataflows.

Time (sec)	#	Min	Max	Mean	Stdev
Montage	100	3.82	49.32	11.32	2.95
Ligo	100	4.03	689.39	222.33	241.42
Cybershake	100	0.55	199.43	22.97	25.08
Input (MB)	**#**	**Min**	**Max**	**Mean**	**Stdev**
Montage	20	0.01	4.02	3.22	1.65
Ligo	53	0.86	14.91	14.24	2.70
Cybershake	52	1.81	19169.75	1459.08	5091.69

Table 5: Indexes on table lineitem.

Column	Type	Index Size	% Table Size
comment	text	422.30 MB	30.16 %
shipinstruct	20 chars	248.95 MB	17.78 %
commitdate	date	225.91 MB	16.13 %
orderkey	integer	146.99 MB	10.49 %

Synthetic data of three real scientific applications, namely Montage [28], Ligo [19] and Cybershake [17], are used to evaluate the proposed approach. Montage shown in Figure 5A is used to generate image mosaics of the sky, LIGO shown in Figure 5B is used to analyze galactic binary systems and Cybershake shown in Figure 5C is used for the characterization of earthquakes. The dataflows are produced using the generator in [8] which specifies the execution time of each operator, the dependencies between them and the sizes of the input/output files of each operator. The basic statistics of the operators are shown in Table 4.

The input files of the dataflows shown in Table 4 are used as a database of files. The total number of files is 125 and their total size is 76.69 GB. The maximum size of a file partition is set equal to 128 MB, resulting in a total number of 713 file partitions. The TPC-H benchmark [1] is used to compute the sizes of typical indexes and model the speed-ups they provide. Table `lineitem` with scale 2 which has approximately 12 million rows and a size of 1.4 GB is used. Table 5 shows the sizes of indexes on four different columns of the table. To model the speed-up that indexes offer, the following SQL queries were created based on the categories presented in Section 1:

Order by:

```
SELECT orderkey FROM lineitem
ORDER BY orderkey;
```

Select range (large):

```
SELECT orderkey FROM lineitem
WHERE orderkey > 1000000
  AND orderkey < 2000000;
```

Select range (small):

Table 6: Index speedup.

Query	No-Index	Index	Speedup
Order by	44.730 sec	6.010 sec	7.44x
Select range (large)	5.103 sec	0.054 sec	94.44x
Select range (small)	4.921 sec	0.016 sec	307.50x
Lookup	4.393 sec	0.007 sec	627.14x

```
SELECT orderkey FROM lineitem
WHERE orderkey > 10000
  AND orderkey < 20000;
```

Lookup:

```
SELECT orderkey FROM lineitem
where orderkey = 1000000;
```

Table 6 shows the speed-up the index on column `orderkey` offers. Four potential indexes for each file are used. Each index size is computed using the percentages shown in Table 5 and its speed-up is randomly chosen from the values of Table 6.

A *Dataflow Generator Client* issues dataflows at time points that follow a Poisson distribution. More specifically, the generator implemented computes the arrival time k (in seconds) of the next dataflow as $f(k;\lambda) = \Pr(X = k) = \lambda^k e^{-\lambda}/k!$, with λ equal to 60 seconds. Dataflows are generated using two settings: randomly (*random generator*) and with phases (*phase generator*). The phase generator produces dataflows to measure the adaptability of the proposed approach to workload changes as follows: Cybershake dataflows for 33.3 quanta (10000 sec), Ligo dataflows for 16.6 quanta (5000 sec), Montage dataflows for 66.6 quanta (20000 sec) and Cybershake dataflows for 27.3 quanta (8200 sec) with each generated dataflow having different speed-ups for the indexes it uses.

6.2 Scheduler robustness for estimation errors

In reality, operator runtimes and data sizes may be overestimated or underestimated. In the first set of experiments, the sensitivity of the scheduler to estimation errors is investigated. To do so, the runtime of operators and the data sizes they generate are randomly varied within a certain percentage and the difference between the actual and estimated values for time, money and fragmentation are computed. For example, for an estimation error of 10% a random value in the range of [90 - 110] seconds is selected to modify the runtime of an operator initially estimated at 100 seconds. Figure 6 shows the results for different values of estimation errors added on the CPU time (operator runtime) and data used. As can be seen, the estimations are robust considering that an error of more than 20% in operator runtime and datasize estimations is relatively high. When the estimations are extremely poor (large errors), the performance of the algorithm can be significantly affected. This is because the algorithm makes scheduling decisions offline (before dataflow execution) based on estimations of operator runtimes and datasizes and does not adapt to unpredicted changes. Future work could investigate how to incorporate estimation errors on decision making to account for inaccurate estimates and yield better performance.

6.3 Comparison of dataflow schedulers

In this set of experiments, the skyline dataflow scheduler proposed (*offline*) is compared with the online load balance scheduler typically used in IaaS clouds (*online*). Operator runtimes and data

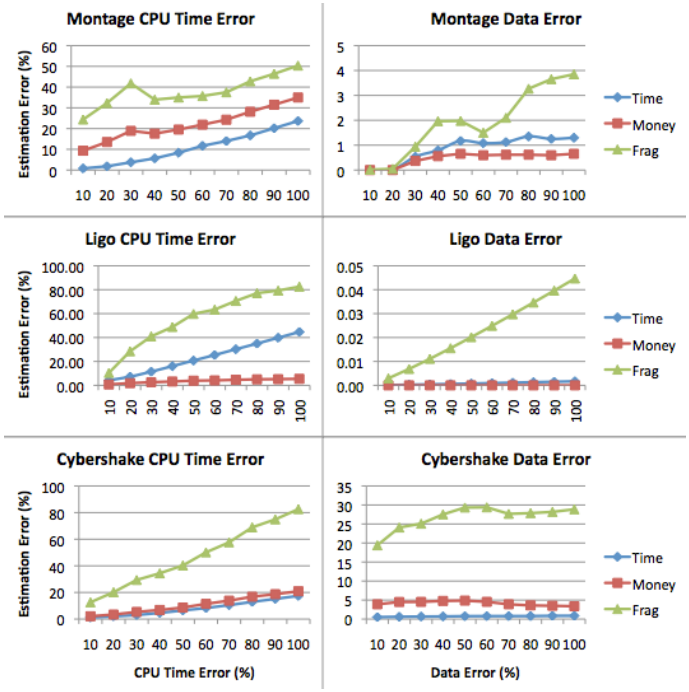


Figure 6: Sensitivity of the offline scheduler to inaccurate estimations.

sizes are scaled to evaluate the efficiency of the proposed scheduler for different scenarios. Since the online scheduler generates a single execution schedule, the fastest schedule from the skyline obtained using the proposed skyline dataflow scheduler (offline) is used for the comparison. The results for Cybershake (the results are similar for the other dataflows) are presented in Figure 7; the y-axis shows the difference (%) between the offline and the online scheduler. The left part of Figure 7 shows the results when scaling the operator runtimes up to 10x (shown in the x-axis) and keeping the data sizes small (scaled to 0.01 of the original size). The online scheduler performs well for these type of dataflows (CPU-intensive) generating faster but slightly more expensive schedules by balancing the load. However, load balancing does not work well for data-intensive dataflows where data placement greatly affects the execution of dataflows. The right part of Figure 7 shows the results when scaling the size of data up to 100x. It can be seen that the schedules generated by the online load balance scheduler are up to 2x slower and up to 4x more expensive compared to the proposed offline scheduler.

6.4 Comparison of index interleaving algorithms

In this experiment, we compare the two index interleaving algorithms proposed; the *LP interleaving* algorithm and the *online* interleaving algorithm. Figure 8 shows the number of indexes built at each schedule in the skylines obtained for Montage using the two index interleaving algorithms (the results are similar for the other two dataflows). The first observation is that the LP interleaving algorithm is able to schedule significantly more build index operators. This is because the information about the fragmented resources is available before the algorithm runs. In contrast, the online algorithm schedules the index build operators and the dataflow operators at the same time. Also, the two skylines obtained are not the same (as can be seen from the monetary cost that corresponds to each point). This is because the

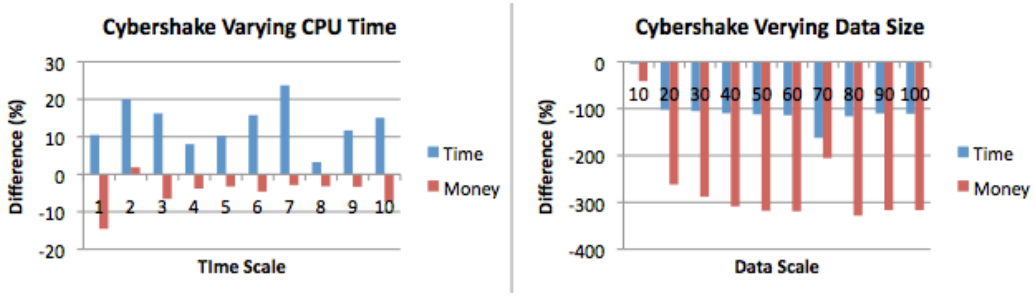


Figure 7: Comparison of the *online* and *offline* scheduler performance.

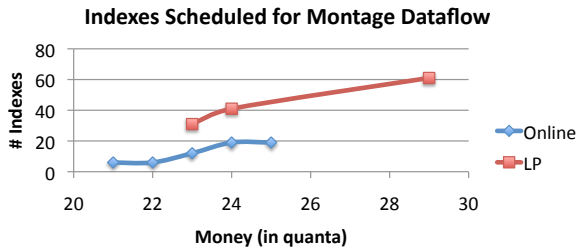


Figure 8: Number of indexes scheduled using different algorithms for Montage dataflow.

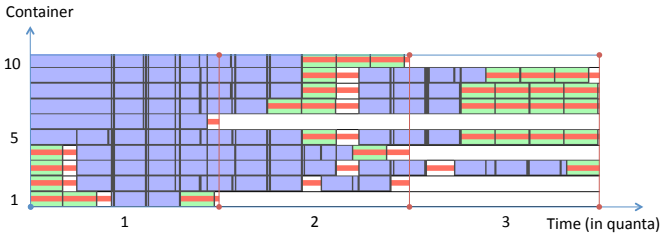


Figure 9: Montage with build index ops (green).

online algorithm interferes with the scheduling of the dataflow operators resulting in cheaper schedules.

Figure 9 shows an example with the timeline of Montage interleaved with build index operators scheduled by the LP interleaving algorithm. Dataflow operators are shown in blue and build index operators are shown in green. The red line indicates idle compute resources. We observe that the LP interleaving algorithm uses a significant amount of idle compute time. The initial idle time is 7.14 quanta and after the assignment of the build index operators, the fragmentation is reduced to 1.6 quanta.

We also compute an upper bound of the quality of the solution found by the LP algorithm by merging all the individual idle time periods and solving the knapsack problem using only one large continuous time segment. We do this using the example of Figure 10, which shows the times of the build index operators and the fragmented resources we used. For simplicity, we set the gain of each operator to be equal to its execution time. As a baseline, we compare with the following greedy algorithm (inspired by Graham [25]): first, we order the operators by descending execution times (and gain in this case) and proceed by assigning each operator to the idle time segment with the most remaining time. A build index operator that does not fit anywhere is not scheduled. Figure 11 shows the results of the LP interleaving algorithm compared to the baseline and the upper bound. We observe that the LP interleaving algorithm is able to find a solution close to the theoretical upper bound (within 5% in this experiment).

6.5 Dynamic Dataflow Workload

In this experiment, the efficiency of the proposed auto-tuning approach (shown in Algorithm 2) is evaluated and compared using the *no-indexes* and *random* approaches as baseline algorithms.

6.5.1 Dataflow Generator with Phases. Initially, the results obtained using the dataflow generator client with phases are presented. Figure 12 shows the number of dataflows finished after 720 time quanta using the different approaches. It can be seen that the number of dataflows executed is doubled when using the proposed approach compared to the baseline where no index is used. Furthermore, the monetary cost spent per dataflow is significantly reduced. It can also be seen that the random approach does not greatly affect the number of finished dataflows compared to the scenario of not using indexes (no index). However the average monetary cost per dataflow is significantly increased due to the storage cost required, which is not taken into account. Finally, the cost per dataflow is increased when non beneficial indexes are maintained, as can be seen by comparing the columns labelled as Gain (no delete) and Gain.

Table 7 shows the total number of operators executed and stopped due to quantum expiration or preemption for the execution of a dataflow operator. It can be seen that the packing achieved by the LP interleaving algorithm is better compared to the random algorithm and fewer build index operators are stopped prematurely.

Table 7: Operators executed.

Algorithm	Total Ops	Killed Ops	Percentage
No Index	22402	0	0
Random	25649	1143	4.4
Gain	49549	1418	2.8

Figure 13 shows the number of indexes built and the total storage cost over time. It can be seen that the proposed approach adapts to the workload by creating and deleting indexes when they become non-beneficial. When Cybershake is re-issued in the final phase, some previously deleted indexes become beneficial again and are recreated.

6.5.2 Random Dataflow Generator. In this experiment, a random dataflow generator client is used. Figure 14 shows the number of dataflows finished after 720 time quanta. The number of dataflows executed is larger using the proposed approach. This is because the average execution time per dataflow is reduced.

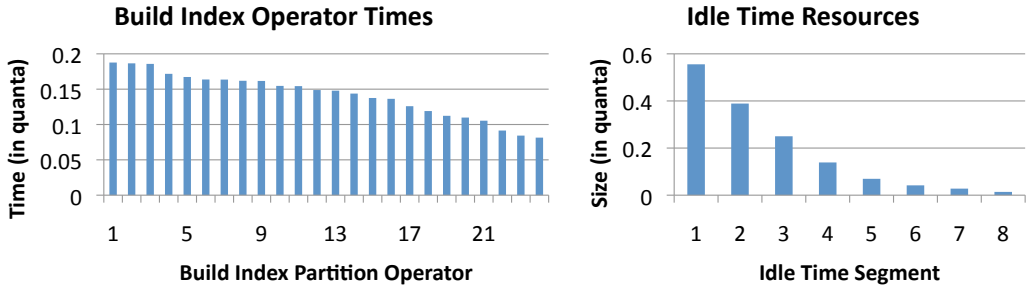


Figure 10: Histogram with execution times of build index operators and idle time resources.

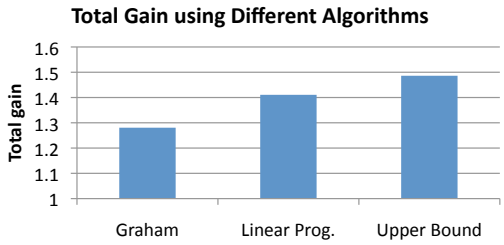


Figure 11: Total gain using different algorithms using the build index operators and idle compute times of Figure 10.

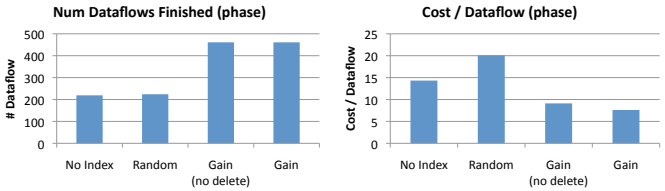


Figure 12: Executed dataflows and average cost/dataflow (phase dataflow generator).

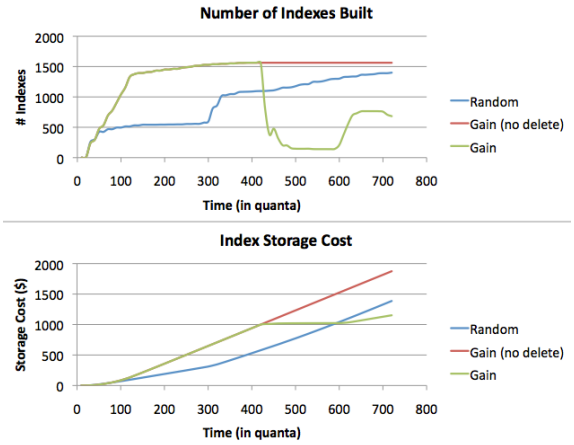


Figure 13: Adaptation of the algorithm to the dataflow workload.

Also, the cost per dataflow is reduced, but not as much as in the previous experiment where the phase dataflow generator client was used. This is because the input is totally at random and, as a result, indexes are stored for a longer period (essentially, they never become non-beneficial). Even in this case, the proposed approach outperforms the baseline approaches.

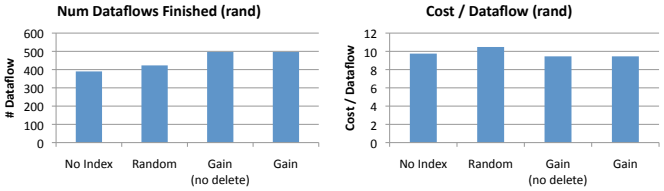


Figure 14: Executed dataflows and average cost per dataflow (random dataflow generator).

7 CONCLUSIONS

In this paper the problem of index management to improve the performance of data-intensive flows on the Cloud is considered. An online auto-tuning approach to assess the usefulness of indexes for the execution of dataflows and utilize idle slots in the execution schedule to build a proper set of indexes is described. The results show that the proposed approach can significantly reduce the average execution time and monetary cost required per dataflow. Future work could evaluate the benefits of index management for scenarios with heterogeneous cloud resources. Also, in this work, we consider a conservative approach to build indexes using idle slots so that they do not interfere with the user workload. Building indexes in a delayed manner for scenarios were idle slots are short is an interesting direction of our future work. Finally, automatic learning of the index gain fading controller to select proper respective values for each index and improve the performance of the proposed approach is another research direction.

ACKNOWLEDGMENT

This work is partially supported by the European Commission under grant agreement 318338, project Optique.

REFERENCES

[1] [n. d.]. TPC-H. http://www.tpc.org/tpch/
[2] 2009. Multi-Version Concurrency Control Algorithms. In *Encyclopedia of Database Systems*. 1870.
[3] Amazon. [n. d.]. Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/
[4] Amazon. [n. d.]. Elastic Map Reduce. http://aws.amazon.com/elasticmapreduce/
[5] Amazon. [n. d.]. Simple Storage Service (S3). http://aws.amazon.com/s3/
[6] Amazon. [n. d.]. Web Services. http://aws.amazon.com/
[7] Apache. [n. d.]. Hadoop. http://hadoop.apache.org/
[8] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, Mei-Hui Su, and K. Vahi. 2008. Characterization of scientific workflows, In WORKS. *"Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on"*, 1 –10. https://doi.org/10.1109/WORKS.2008.4723958
[9] N. Bruno and S. Chaudhuri. 2007. An Online Approach to Physical Design Tuning. In *ICDE*. 826–835. https://doi.org/10.1109/ICDE.2007.367928
[10] Nicolas Bruno and Surajit Chaudhuri. 2010. Constrained physical design tuning. *VLDB J.* 19, 1 (2010), 21–44.

[11] Gang Chen, Hoang Tam Vo, Sai Wu, Beng Chin Ooi, and M. Tamer Özsu. 2011. A Framework for Supporting DBMS-like Indexes in the Cloud. *PVLDB* 4, 11 (2011), 702–713.

[12] Y. Chronis et al. 2016. A relational approach to complex dataflowss. In *EDBT/ICDT Workshops*.

[13] Mariano P. Consens, Kleoni Ioannidou, Jeff LeFevre, and Neoklis Polyzotis. 2012. Divergent physical design tuning for replicated databases. In *SIGMOD Conference*. 49–60.

[14] Carlo Curino, Evan P. C. Jones, Samuel Madden, and Hari Balakrishnan. 2011. Workload-aware database monitoring and consolidation. In *SIGMOD Conference*. 313–324.

[15] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. ACM, New York, NY, USA, 666–679. https://doi.org/10.1145/3299869.3314035

[16] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. 2011. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. *PVLDB* 4, 6 (2011), 362–372.

[17] Ewa Deelman et al. 2006. Managing Large-Scale Workflow Execution from Resource Provisioning to Provenance Tracking: The CyberShake Example. In *e-Science*. 14. https://doi.org/10.1109/E-SCIENCE.2006.99

[18] Ewa Deelman and Ann L. Chervenak. 2008. Data Management Challenges of Data-Intensive Scientific Workflows. In *CCGRID*.

[19] Ewa Deelman, Carl Kesselman, et al. 2002. GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists. In *HPDC*. 225. https://doi.org/10.1109/HPDC.2002.1029922

[20] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. 2012. Only Aggressive Elephants are Fast Elephants. *PVLDB* 5, 11 (2012), 1591–1602.

[21] Iman Elghandour and Ashraf Aboulnaga. 2012. ReStore: Reusing Results of MapReduce Jobs. *Proc. VLDB Endow.* 5, 6 (Feb. 2012), 586–597.

[22] Wenfei Fan, Floris Geerts, and Frank Neven. 2013. Making Queries Tractable on Big Data with Preprocessing. *PVLDB* 6, 9 (2013).

[23] Daniela Florescu and Donald Kossmann. 2009. Rethinking cost and performance of database systems. *SIGMOD Record* 38, 1 (2009), 43–48.

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *SOSP*. 29–43.

[25] Ronald L. Graham. 1969. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal of Applied Mathematics* 17, 2 (1969), 416–429.

[26] Ashish Gupta, Fan Yang, Jason Govig, et al. 2014. Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing. *PVLDB* 7, 12 (2014), 1259–1270.

[27] Ramanujam Halasipuram, Prasad M. Deshpande, and Sriram Padmanabhan. 2014. Determining Essential Statistics for Cost Based Optimization of an ETL Workflow. In *EDBT*. 307–318.

[28] Joseph C. Jacob et al. 2009. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *IJCSE* 4, 2 (2009), 73–87. https://doi.org/10.1504/IJCSE.2009.026999

[29] Ivo Jimenez, Huascar Sanchez, Quoc Trung Tran, and Neoklis Polyzotis. 2012. Kaizen: a semi-automatic index advisor. In *SIGMOD Conference*. 685–688.

[30] Verena Kantere, Debabrata Dash, Georgios Gratsias, and Anastasia Ailamaki. 2011. Predicting cost amortization for query services. In *SIGMOD Conference*. 325–336.

[31] Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. *Knapsack problems*. Springer. I–XX, 1–546 pages.

[32] Herald Kllapi, Boulos Harb, and Cong Yu. 2014. Near neighbor join. In *ICDE*. 1120–1131.

[33] Herald Kllapi, Eva Sitaridi, Manolis M. Tsangaris, and Yannis E. Ioannidis. 2011. Schedule optimization for data processing flows on the cloud. In *Proc. of SIGMOD*. 289–300.

[34] Donald Kossmann. 2000. The State of the art in distributed query processing. *Comput. Surveys* 32, 4 (2000), 422–469.

[35] Avinash Lakshman and Prashant Malik. 2009. Cassandra: structured storage system on a P2P network. In *PODC*. 5.

[36] Jeff LeFevre et al. 2014. MISO: souping up big data query processing with a multistore system. In *SIGMOD Conference*. 1591–1602.

[37] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. 2012. Robust Estimation of Resource Consumption for SQL Queries using Statistical Techniques. *PVLDB* 5, 11 (2012).

[38] Tanu Malik, Xiaodan Wang, Debabrata Dash, Amitabh Chaudhary, Anastasia Ailamaki, and Randal C. Burns. 2009. Adaptive Physical Design for Curated Archives. In *SSDBM*. 148–166.

[39] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*.

[40] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. 2005. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming* 13, 4 (2005), 277–298.

[41] Stefan Richter, Jorge-Arnulfo Quiané-Ruiz, Stefan Schuh, and Jens Dittrich. 2014. Towards zero-overhead static and adaptive indexing in Hadoop. *VLDB J.* 23, 3 (2014), 469–494.

[42] R. Schlosser, J. Kossmann, and M. Boissier. 2019. Efficient Scalable Multi-attribute Index Selection Using Recursive Strategies. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1238–1249. https://doi.org/10.1109/ICDE.2019.00113

[43] Karl Schnaitter and Neoklis Polyzotis. 2012. Semi-Automatic Index Tuning: Keeping DBAs in the Loop. *PVLDB* 5, 5 (2012).

[44] Karl Schnaitter, Neoklis Polyzotis, and Lise Getoor. 2009. Index Interactions in Physical Design Tuning: Modeling, Analysis, and Applications. *PVLDB* 2, 1 (2009), 1234–1245.

[45] Alkis Simitsis. 2003. Modeling and managing ETL processes. In *VLDB PhD Workshop*.

[46] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*. 996–1005.

[47] Quoc Trung Tran, Ivo Jimenez, Rui Wang, Neoklis Polyzotis, and Anastasia Ailamaki. 2013. RITA: An Index-Tuning Advisor for Replicated Databases. *CoRR* abs/1304.1411 (2013).

[48] Manolis M. Tsangaris et al. 2009. Dataflow Processing and Optimization on Grid and Cloud Infrastructures. *IEEE Data Eng. Bull.* 32, 1 (2009), 67–74.

[49] Prasang Upadhyaya, Magdalena Balazinska, and Dan Suciu. 2012. How to Price Shared Optimizations in the Cloud. *PVLDB* 5, 6 (2012).

[50] Rui Wang, Quoc Trung Tran, Ivo Jimenez, and Neoklis Polyzotis. 2013. INUM+: A leaner, more accurate and more efficient fast what-if optimizer. In *ICDE Workshops*. 50–55.

[51] Petrie Wong, Zhian He, and Eric Lo. 2013. Parallel analytics as a service. In *SIGMOD Conference*. 25–36.

[52] Eugene Wu and Samuel Madden. 2011. Partitioning techniques for fine-grained indexing. In *ICDE*. 1127–1138.

[53] Q. Zhu and T. Tung. 2012. A Performance Interference Model for Managing Consolidated Workloads in QoS-Aware Clouds. In *Proceedings of the 5th IEEE CLOUD*. IEEE, 170–179.