RIPPLE: A Scalable Framework for Distributed Processing of Rank Queries

George Tsatsanifos
National Technical University of Athens
Athens, Greece
gtsat@dblab.ece.ntua.gr

Dimitris Sacharidis
Institute for the Management of Information Systems
Athens, Greece
dsachar@imis.athena-innovation.gr

Timos Sellis
RMIT University
Melbourne, Australia
timos.sellis@rmit.edu.au

ABSTRACT

We introduce a generic framework, termed RIPPLE, for processing rank queries in decentralized systems. Rank queries are particularly challenging, since the search area (i.e., which tuples qualify) cannot be determined by any peer individually. While our proposed framework is generic enough to apply to all decentralized structured systems, we show that when coupled with a particular distributed hash table (DHT) topology, it offers guaranteed worst-case performance. Specifically, rank query processing in our framework exhibits tunable polylogarithmic latency, in terms of the network size. Additionally we provide a means to trade-off latency for communication and processing cost. As a proof of concept, we apply RIPPLE for top-k query processing. Then, we consider skyline queries, and demonstrate that our framework results in a method that has better latency and lower overall communication cost than existing approaches over DHTs. Finally, we provide a RIPPLE-based approach for constructing a k-diversified set, which, to the best of our knowledge, is the first distributed solution for this problem. Extensive experiments with real and synthetic datasets validate the effectiveness of our framework.

1. INTRODUCTION

The term *rank queries* refers to queries that enforce an order on tuples and usually request a few of the highest ranked tuples. We consider three types of rank queries. *Top-k queries* [9] is the simplest, imposing a weak order on the domain via a monotonic function (a weak order is essentially ranking with ties). The answer of a top-k query is a set of k tuples that have the highest score among all other possible k-sets.

Skyline queries [3] impose a partial order on the domain defined by the Pareto aggregation of (total or partial) orders specified on each attribute individually (in a partial order, two domain values may be incomparable). The answer of a skyline query is the set of maximal tuples under this partial order, termed the *skyline*. Note that while, in the weak order of top-k queries, there exists only one domain value for which no tuple with better value exists, in the partial order of skyline queries, there can be multiple domain values for which no tuple with better value exists.

The *k-diversification query* [5] reconciles two conflicting notions. The *relevance* of a tuple is defined by its distance to a given query tuple. On the other hand, the *diversity* of a tuple with respect to a set of tuples is determined by its aggregate distance to these tuples. The answer of a k-diversification query is a set of k tuples that takes the highest value in an *objective function* combining the relevance and diversity of its tuples. Note that, in k-diversification, sets of tuples, rather than individual tuples, are ranked; hence the problem is NP-hard [7].

Our work deals with the distributed evaluation of rank queries in *structured decentralized systems*. In these systems, e.g., [13, 15], the individual servers, termed *peers*, are organized in a content-aware manner, implementing a *distributed hash table* (DHT). Each tuple and each participating peer is assigned a point (a key) from the same domain. A peer becomes responsible for a range of the domain, and stores all tuples falling in this range. Therefore, when searching for a particular tuple, the responsible peer can be easily identified, i.e., by means of looking up the DHT.

Rank queries, in general, are particularly challenging for distributed processing. The reason is that peers have only partial knowledge of the data distribution, and thus no single peer alone can know where qualifying tuples may reside beforehand, i.e., when the query is posed. In other words, the search area is initially unbounded and becomes progressively refined while qualifying tuples are being retrieved. Contrast this to range queries, which request all objects within a particular range, say within distance r around a given point. In the case of a range query, the search area is explicitly defined in the query.

Since the search area is unbounded, there exists a straightforward approach for distributed processing a rank query in any DHT: broadcast the query to the entire network, collect all locally qualifying tuples, and finally derive the answer from them. This method has only a single advantage, in that worst-case network latency is optimal. In the worst case, when the initiating peer and a peer holding an answer tuple are as remote as possible, the latency equals the network diameter, i.e., the maximum number of hops in the shortest path between peers.

Of course, naïve processing has several drawbacks. First, all peers are reached independently of the query and whether they possess contributing tuples to the answer. Second, the communication overhead is very high as many tuples have to be transmitted over the network since it is not possible to locally prune them. For example, in the case of a top-k query, using only local knowledge, each peer must transmit k tuples. Third, the processing cost at the initiating peer is huge, exactly because a large number of tuples is retrieved, and a large number of nodes is encountered which otherwise could have been prevented.

This work proposes *RIPPLE*, a generic scalable framework with tunable latency for processing rank queries in DHTs. The princi-

EDBT 2014, March 24-28, 2014, Athens, Greece.

ple idea of RIPPLE is to exploit the local information within each peer regarding the distribution of tuples and neighboring peers in the domain. Each peer partitions the entire domain into *regions* and assigns them to its neighbors. Then, it prioritizes the forwarding of requests to its neighbors by taking into account the current *state* of query processing, as derived from its own request and from answers collected from remote peers. A single parameter in RIPPLE trades off latency for communication overhead. We emphasize that RIPPLE can be implemented on top of any DHT. However, when paired with MIDAS [16], an inherently multidimensional index based on the k-d tree, RIPPLE exhibits polylogarithmic latency in terms of the network size.

We first apply the RIPPLE framework for top-k queries. Then, we turn our attention to distributed skyline processing using RIPPLE. For the case when RIPPLE is implemented over MIDAS, we also propose an optimization of the index structure with significant performance gains. The resulting approach is shown to have lower latency and/or cause less congestion, depending on the tune parameter, compared to state-of-the-art methods for skyline processing over DHTs. Finally, we instantiate RIPPLE for k-diversification queries. To the best of our knowledge, ours is the first work to address this type of query in a distributed setting. Initially, we use RIPPLE to solve the simpler sub-problem of finding the best tuple to append to a set of $k-1$ tuples. Then, we propose a heuristic solution for answering k-diversification queries. An extensive experimental simulation using real and synthetic datasets demonstrates the key features of RIPPLE: tunable latency and low communication overhead for processing rank queries.

The remainder of this paper is organized as follows. Section 2 discusses related work on distributed processing of rank queries. Section 3 details the RIPPLE framework. Then, Sections 4, 5 and 6 present the application of RIPPLE for the case of top-k, skyline, and k-diversification queries, respectively. Section 7 presents a thorough experimental evaluation of our framework, and Section 8 concludes our work.

2. RELATED WORK

Sections 2.1 and 2.2 review related work on distributed top-k and skyline processing, respectively. Section 2.3 overviews the MIDAS distributed index.

2.1 Distributed Top-k Processing

Top-k processing involves finding the k tuples which are ranked higher according to some ranking function. We distinguish two variants of the distributed version of the problem. In the *vertically distributed* setting, a peer maintains all tuples but stores the values on a single attribute. The seminal work of [6], was the first to address this problem, and introduces the famous Threshold Algorithm (TA) and Fagin's Algorithm (FA). Subsequent works attempt to improve this result. In [4], the Three-Phase Uniform Threshold (TPUT) algorithm is proposed, which in substance improves limitations of the TA. Later, TPUT was improved by KLEE [11], which also supports approximate top-k retrieval, and comes in two flavors, one that requires three phases, and another that needs two round-trips.

The second variant is for *horizontally distributed* data, which is the setting considered in our work. In this case, a peer maintains only a subset of all tuples, but stores all their attributes. There exists significant work for unstructured peer-to-peer networks. A flooding-like algorithm followed by a merging phase is proposed in [1]. In [2], super-peers are burdened with resolving top-k queries, an approach which imposes high execution skew. In SPEERTO [17] each node computes its k-skyband as a pre-processing step. Then, each super-peer aggregates the k-skyband sets of its nodes to answer incoming queries. BRANCA [21] and ARTO [14] cache previous final and intermediate results to avoid recomputing parts of new queries. To the best of our knowledge, no work considers the case of horizontally distributed data over structured overlays, which is the topic of our paper.

2.2 Distributed Skyline Computation

The skyline query retrieves the tuples for which there exists no other tuple that is better on all dimensions. A complete survey on distributed skyline computation can be found in [8], where methods for structured and unstructured networks are thoroughly studied. Regarding structured peer-to-peer networks, which is the setting in our work, DSL [20] leverages CAN [13] for indexing multidimensional data. During query processing, DSL builds a multicast hierarchy in which the peer that is responsible for the region containing the lower-left corner of the constraint is the root. The hierarchy is built in such a way that only peers whose data points cannot dominate each other are queried in parallel. A peer that receives a query along with the local result set, first waits to receive the local skyline sets from all neighboring peers that precede it in the hierarchy. Then, it computes the skyline set based on its local data and the received data points. Thereafter, the local skyline points are forwarded to the peers responsible for neighboring regions, in such a way that only peers whose data points cannot dominate each other are queried in parallel. Besides, neighboring peers that are dominated by the local skyline points are not queried because they cannot contribute to the global skyline set.

Wang et al. present in [18] SSP (Skyline Space Partitioning) for distributed processing of skyline queries in BATON [10]. The multi-dimensional data space is mapped to unidimensional keys using a Z-curve, due to BATON limitations. Query processing starts only at the peer responsible for the region containing the origin of the data space. It computes the local skyline points that are in the global skyline set, and next, it selects the most dominating point used to refine the search space and to prune dominated peers. The querying peer forwards the query to the peers that are not pruned and gathers their skyline sets. Skyframe [19] is applicable for BATON and CAN networks. In Skyframe the querying peer forwards the query to a set of peers called border peers. A peer that is responsible for a region with minimum value in at least one dimension is called border peer. Once the initiator receives the local skyline results, it determines if additional peers need to be queried. Then, the querying peer queries additional peers, if necessary, and gathers the local skyline results. When no further peers need to be queried, the query initiator computes the global skyline set.

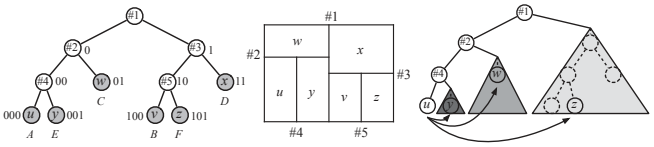
2.3 The MIDAS Overlay

The organization of peers in MIDAS is based on a virtual k-d tree, indexing a d-dimensional domain [16]. The k-d tree is a binary tree, in which each node corresponds to an axis parallel rectangle; the root corresponds to the entire domain. Each internal node has always two children, whose rectangles are derived by splitting the parent rectangle at some value along some dimension, decided by MIDAS.

Each node of the k-d tree is associated with a binary identifier corresponding to its path from the root, which is defined recursively. The root has the empty id $\varnothing$; the left (resp. right) child of an internal node has the id of its parent augmented with 0 (resp. 1). Figure 1(a) shows a virtual k-d tree, and labels the ids of the peers and the internal nodes.

A peer in MIDAS corresponds to a *leaf* of the k-d tree, and stores all tuples, who fall in the leaf's rectangle, which is called its *zone*. Figure 1(b) shows the zones of the peers. Therefore, the size of

the overlay equals the number of leaves in the virtual k-d tree. A MIDAS peer maintains a list of links to other peers. In particular its i-th link points to some peer within the sibling subtree rooted at depth i. Figure 1 shows the links of peer u. It is shown that the expected depth of the MIDAS virtual k-d tree, which determines the diameter of MIDAS, for an overlay of n peers is $O(\log n)$ [16].



(a) The virtual k-d tree (b) Peer zones (c) Links of u

Figure 1: An example of a two-dimensional MIDAS overlay.

3. THE RIPPLE FRAMEWORK

Section 3.1 describes the generic RIPPLE framework, while Section 3.2 focuses on the RIPPLE implementation over MIDAS.

3.1 Generic RIPPLE

We present the RIPPLE generic framework for distributed processing rank queries. We make little assumptions on the underlying DHT. For a peer w, we denote as w.link the set of its neighbors to which w maintains links. The number of w's neighbors is denoted by $|w.\text{link}|$; the maximum number of neighbors in any peer is denoted as Δ.

A key notion in RIPPLE is that of the *region*. RIPPLE associates with each neighbor of w a region, denoted as $w.\text{link}[i].\text{region}$. The region of all w neighbors form a partition of the entire domain. There is an important distinction between the notions of *region* and *zone*. Recall that in DHTs, a peer is assigned a sub-area of the domain, termed the zone and stores all tuples within its zone. On the other hand, the region of a neighbor (a RIPPLE-only notion) is generally a much larger area, which however encompasses the zone. More importantly, a region depends on the viewpoint of a specific peer, and thus a peer might be associated with different regions. For example, consider peers w, v who both have peer x as their neighbor. The region of x from w's viewpoint can be different from the region of x from u's viewpoint; however both regions contain the zone of x.

Depending on the underlying DHT, there is often a natural way to assign regions to the neighbors of a peer w. A region should satisfy two requirements: (i) a region should cover the peer's zone, and (ii) the union of the regions of a peer's neighbors should form a partition covering the domain. We next discuss how to define regions for three conspicuous DHTs; the definition to other types of overlays is straightforward.

In CAN, each peer w has at least two neighbors along each dimension. More specifically, in a d-dimensional domain, two nodes are neighbors if their coordinate spans overlap along $d-1$ dimensions and abut along one. Hence, the lower (resp. upper) neighbor along the i-th dimension represents a region that resembles a pyramidal frustum (a trapezoid in 2-d; a pyramid whose top has been cut-off in higher dimensions) having as base the lower (resp. upper) boundary face of the domain that is also perpendicular to the i-th dimension, and as top the lower (resp. upper) face of w's zone, which is perpendicular to the i-th dimension. Thereby, a peer will forward a query that either receives or issues to the node(s) whose region(s) overlap with the query.

In Chord, each peer w has neighbors whose zones cover domain points at exponentially increasing distances from w. Then, the region of w's i-th neighbor is defined as the area of the domain stretching from the beginning of the i-th neighbor zone until the beginning of the $(i+1)$ neighbor zone (or w's zone if i-th is the last neighbor).

In MIDAS, each peer w has a neighbor inside each sibling subtree rooted at depth up to w.depth. Then, the region of w's i-th neighbor is defined as the area of the domain covered by the sibling subtree rooted at depth i.

Regarding query processing, we use Q to abstractly refer to the query. We denote as A the *local answer*, i.e., the local tuples that satisfy the query. Query processing begins at the initiator peer, which we denote as v. Each peer, including the initiator, that is involved in query processing executes the same procedure and returns its local qualifying tuples to the initiator. Depending on the query, the initiator might have to perform additional operations in order to extract the final answer.

A key concept in RIPPLE is that of the *state*, denoted as S, which consists of a (partial) view of the distributed query processing progress. For example, depending on the query and the distributed algorithm, S could be a set of local/remote records, or bounds/guarantees for these tuples. We distinguish between two types of state. The *local state* at peer w, denoted as S_w^L, contains only information collected at w, both from local tuples and from remote states which w has explicitly requested. The global state at peer w, denoted as S_w^G, encompasses the local state S_w^L and also includes information that was forwarded to w together with the query.

The basic idea of the RIPPLE framework is to exploit regions and states, acquiring knowledge regarding the progress of query processing, in order to meticulously guide the search to its neighbor peers. Before presenting RIPPLE, we first describe two extreme settings. The first is called fast and optimizes for latency.

Algorithm 1 shows fast query processing at each peer. A peer w receives the query Q, a global state S^G, the address of the initiator v, and the restriction area R within which query processing should be confined. The restriction area ensures that no peer will receive the same request twice. We emphasize that all algorithms in this section are *templates* and contain calls to abstract functions, whose operations depend on the exact query type, and which are elaborated in the following sections.

Algorithm 1 w.fast(v, Q, S^G, R) processes query Q, initiated by v and with current global state S^G, within area R.

```
1: S_w^L ← w.computeLocalState(Q, S^G)
2: S_w^G ← w.computeGlobalState(Q, S^G, S_w^L)
3: for each link i do
4:     if w.isLinkRelevant(i, Q, S_w^G, R) then
5:         w.link[i].fast(v, Q, S_w^G, w.link[i].region∩R)
6:     end if
7: end for
8: A ← w.computeLocalAnswer(Q, S_w^L)
9: w.sendLocalAnswerTo(v, A)
```

Based on the received global state S^G and the local tuples, peer w computes its local state by invoking computeLocalState. Also, it computes its global state by invoking computeGlobalState (line 2). Then, w considers all its neighbors in turn (lines 3–7). Subsequently, peer w invokes isLinkRelevant (line 4) to check whether the region of the i-th neighbor (1) overlaps with the restriction area R, and (2) contains tuples that can contribute to the answer, given the global state S_w^G. If the i-th neighbor passes the check, the query is forwarded to it, along with the global state and the restriction area set to the intersection of R with the i-th region (line 5). After considering all neighbors, peer w computes the local answer based on its local state, invoking computeLocalAnswer (line 8), and sends only its local qualifying tuples to the initiator v (line 9).

If Algorithm 1 is initially invoked with restriction area equal to

the entire domain, then it correctly processes query Q, subject to the abstract functions being correct. To understand this, observe that if we ignore the second check of isLinkRelevant (whether a neighbor contains local tuples based on the local state), then all peers in the network will be reached exactly once. The maximum latency of fast is equal to the diameter of the network, as all neighbors are contacted at once.

Algorithm 1 optimizes latency, and tries to reduce the communication cost as much as possible. In the following, we present the second extreme setting of RIPPLE, termed slow, which optimizes the communication cost at the expense of latency. Algorithm 2 shows query processing at each peer. As before, the algorithm restricts query processing in sub-areas of the domain and employs local states. The difference is that query propagation is performed iteratively, and local state is updated after each iteration. The rationale is that the communication cost depends on the information derived locally in the peers (i.e., from the local states). Ideally, but unfeasibly, the communication cost is minimized when each peer has complete knowledge of all tuples stored in the network.

In slow, a peer w receives the query Q, the current global state S^G, the address of the initiator v, the address of the peer u that sent this message, and the restriction area R. Initially, it computes its local state based on the received global state (line 1), and then its global state (line 2). The next steps differ from Algorithm 1. Peer w prioritizes its neighbors according to their potential contribution to the query. Function sortLinks sorts the links of w using the function comp, which compares the priorities of the i-th and j-th neighbors (line 3).

Then, slow considers each neighbor in decreasing significance (lines 3–10). Let ℓ-th be the currently examined neighbor. Similarly to fast peer w invokes isLinkRelevant (line 4) to check whether the ℓ-th neighbor should be contacted. If the check returns true, the query is forwarded to this neighbor, along with the global state and the restriction area appropriately set (line 5). In contrast to Algorithm 1 however, w waits for a response from its link. Upon receiving the response (line 6), which includes a remote local state, peer w invokes updateLocalState to encorporate this state to its own local state (line 7). Also, it re-computes the global state taking into account the update local state, by invoking computeGlobalState again (line 8). Then it continues to examine the next neighbor according to the prioritization. As the iterations progress, the local state is continuously enriched with information from neighbors.

After considering all its neighbors, peer w sends its local state to peer u, who forwarded the query to w (line 11). Subsequently, w computes the answer, invoking computeLocalAnswer (line 12), and sends the local qualifying tuples to the initiator v (line 13).

Algorithm 2 w.slow(v, u, Q, S^G, R) processes query Q initiated by v and forwarded by u, with current global state S^G, within area R.

1: $S_w^L \leftarrow w$.computeLocalState(Q, S^G)
2: $S_w^G \leftarrow w$.computeGlobalState(Q, S^G, S_w^L)
3: **for** $\ell \in w$.sortLinks(w.comp(i, j, Q)) **do**
4: **if** w.isLinkRelevant(ℓ, Q, S_w^G, R) **then**
5: w.link[ℓ].slow(v, w, Q, S_w^G, w.link[ℓ].region$\cap R$)
6: $S^L \leftarrow w$.receiveRemoteLocalState()
7: $S_w^L \leftarrow w$.updateLocalState($Q, \{S_w^L, S^L\}$)
8: $S_w^G \leftarrow w$.computeGlobalState(Q, S^G, S_w^L)
9: **end if**
10: **end for**
11: w.sendLocalStateTo(u, S_w^L)
12: $A \leftarrow w$.computeLocalAnswer(Q, S_w^L)
13: w.sendLocalAnswerTo(v, A)

It is easy to see that Algorithm 2 is correct if it is initially invoked with a restriction area equal to the entire domain, and the abstract functions are correct. As before, if we ignore the second check of function isLinkRelevant, all peers in the network will be reached. However, the maximum latency is different. Observe that each peer contacts only one neighbor at a time, waiting for its response. The response comes only after the message is forwarded to all the peers within the restriction area. Since, each subsequent peer follows the same strategy, the waiting time (in number of hops) equals the total number of peers within the restriction area. Therefore, the maximum latency of slow is equal to the network size. Of course, in practice due to the prioritization, slow query processing terminates much sooner, without the need to contact all peers.

We are now ready to present the ripple distributed algorithm, which constitutes the heart of our framework. This algorithm tradeoffs between latency and communication cost via the ripple parameter r. Aiming to minimize communication cost, the ripple algorithm prioritizes the search, meticulously propagating the query to peers that are expected to contribute to the answer, similar to slow. Aiming to control the maximum latency, after the query reaches peers more than r hops away from the initiator, the query begins to propagate in *ripples*, similar to fast. Essentially, the ripple algorithm believes that the first few (prioritized) hops are important in order to construct a good local state as soon as possible, which will then be used to better guide the search.

To enforce the previous reasoning, ripple on peer w mandates each peer reached up to r hops away from w to execute slow, and each peer farther than r hops from w to execute fast. At the extreme case when $r = 0$, ripple degenerates to fast. At the other extreme, when r is sufficiently large (greater than the maximum number of neighbors Δ), ripple degenerates to slow.

Algorithm 3 shows ripple query processing at each peer. A peer w receives the query Q, the current state S, the address of the initiator v, the address of the peer u that sent this message, the restriction area R, and the value of the parameter r. Initially, ripple computes the local state and the global based on the received global state (lines 1–2). Then depending on the value of r, one of two loops is executed. The first loop (lines 4–11) is essentially the main loop of Algorithm 2, with the exception that multiple states might be received (line 7) that need processing, i.e., updating the local state and computing the global state (line 8–9). Also note that the value of the r parameter in the forwarded query is decreased. On the other hand, the second loop (lines 13–17) is essentially the main loop of Algorithm 1; all subsequent peers receive an r value of 0.

At the end of both loops, the local state is sent to the parent of w that forwarded this request, in the case the first loop is executed, or the ancestor peer for $r = 1$ that forwarded this request, in the case the second loop is executed (line 19); in any case, this peer's address u is included in the request. Finally, w computes the answer, invoking computeLocalAnswer (line 18), and sends the local qualifying tuples to the initiator v (line 19).

Algorithm 3 is correct if it is initially invoked with a restriction area equal to the entire domain, and the abstract functions are correct. The worst-case latency of the algorithm depends on the r parameter and the underlying DHT; Section 3.2 analyzes worst-case latency in MIDAS. For low r values the worst-case latency is closer to the network diameter, while for large r values it is closer to the network size.

3.2 Analysis of RIPPLE for MIDAS

This section assumes that the underlying DHT in RIPPLE is MIDAS. In this case the regions and the restriction areas in the algorithms of the previous section are subtrees. Hence the parameter R can be replaced with the depth δ of the subtree in which processing is to be restricted. Then, the worst-case latency can be expressed in terms of δ, as the next lemmas suggest.

Algorithm 3 w.ripple(v, u, Q, S^G, R, r) processes query Q forwarded by u and with current global state S^G, within area R and with ripple parameter value r.

```
1: S_w^L ← w.computeLocalState(Q, S^G)
2: S_w^G ← w.computeGlobalState(Q, S^G, S_w^L)
3: if r > 0 then
4:    for ℓ ∈ w.sortLinks(w.comp(i, j, Q)) do
5:       if w.isLinkRelevant(ℓ, Q, S_w^G) then
6:          w.link[ℓ].ripple(v, w, Q, S_w^G, w.link[ℓ].region∩R, r − 1)
7:          {S_i^L} ← w.receiveRemoteLocalState()
8:          S_w^L ← w.updateLocalState(Q, {S_w^L, {S_i^L}})
9:          S_w^G ← w.computeGlobalState(Q, S^G, S_w^L)
10:      end if
11:   end for
12: else
13:   for each link i do
14:      if w.isLinkRelevant(i, Q) then
15:         w.link[i].ripple(v, u, Q, S_w^G, w.link[ℓ].region∩R, 0)
16:      end if
17:   end for
18: end if
19: w.sendLocalStateTo(u, S_w^L)
20: A ← w.computeLocalAnswer(Q, S_w^L)
21: w.sendLocalAnswerTo(v, A)
```

LEMMA 1. *The worst-case latency of Algorithm 1 for MIDAS is* $L_f(\delta) = \Delta - \delta$.

PROOF. First, observe that $L_f(\Delta) = 0$, as no message needs to be transmitted. At iteration i, Algorithm 1 forwards the query to a link and restricts it to the sibling subtree at depth i. Recursively, this iteration causes worst-case latency of $1 + L_f(i)$. Since, all iterations are executed at once, the worst-case latency is determined by the largest worst-case latency at any sibling subtree. Thus:

$$L_f(\delta) = 1 + \max_{i=\delta+1}^{\Delta} L_f(i).$$

Since $L_f(i) > L_f(i+1)$, we obtain the recursion which solves to:

$$L_f(\delta) = 1 + L_f(\delta + 1) = \Delta - \delta.$$

□

Setting $\delta = 0$, we obtain that the worst-case latency for processing a rank query according to Algorithm 1 is Δ, which is $O(\log n)$ and equals the diameter of MIDAS.

LEMMA 2. *The worst-case latency of Algorithm 2 for MIDAS is* $L_s(\delta) = 2^{\Delta-\delta} - 1$.

PROOF. It holds that $L_s(\Delta) = 0$, and that each iteration at depth ℓ introduces worst-case latency of $1 + L_s(\ell)$. Since the algorithm waits for a response in each iteration before continuing to the next, the total worst-case latency is given by sum of the per-iteration latencies, independently of the order in which sibling subtrees are considered. Therefore:

$$L_s(\delta) = \sum_{\ell=\delta+1}^{\Delta} (1 + L_s(\ell)).$$

From which we obtain the recursion which solves to:

$$L_s(\delta) = 1 + 2 \cdot L_s(\delta + 1) = 2^{\Delta-\delta} - 1.$$

□

Setting $\delta = 0$, we obtain that the worst-case latency for processing a rank query according to Algorithm 2 is 2^{Δ}, which is $O(n)$. However, note that as our experimental analysis shows, due to the prioritization, the average latency of slow is much lower.

Finally, regarding the worst-case latency of the ripple algorithm, the following result holds.

LEMMA 3. *The worst-case latency of Algorithm 3 for MIDAS is given by the recurrence* $L_r(\delta, r) = 1 + L_r(\delta+1, r) + L_r(\delta+1, r-1)$ *with initial conditions* $L_r(\delta, 0) = \Delta - \delta$ *and* $L_r(\Delta, r) = 0$.

PROOF. The first initial condition holds, because, for $r = 0$, Algorithm 3 executes the second loop which is identical to Algorithm 1. The second initial condition holds, because, for $\delta = \Delta$, both loops execute no iteration.

Next consider the case of $r > 0$, when the first loop is executed. Each iteration at depth ℓ introduces worst-case latency of $1 + L_r(\ell, r-1)$. The total worst-case latency is given by sum of the per-iteration latencies:

$$L_r(\delta, r) = \sum_{\ell=\delta+1}^{\Delta} (1 + L_r(\ell, r-1)).$$

Taking the difference $L_r(\delta, r) - L_r(\delta + 1, r)$, we obtain the given recurrence. □

While we could not derive a closed-form formula for the partial recurrence equation of the lemma, we have analytically computed $L_r(\delta, r)$ for various values of r:

$$L_r(\delta, 1) = \frac{1}{2}(\Delta - \delta)^2 + \frac{1}{2}(\Delta - \delta)$$
$$L_r(\delta, 2) = \frac{1}{6}(\Delta - \delta)^3 - \frac{1}{2}(\Delta - \delta)^2 + \frac{4}{3}(\Delta - \delta) - 1$$
$$L_r(\delta, 3) = \frac{1}{24}(\Delta - \delta)^4 - \frac{1}{4}(\Delta - \delta)^3 + \frac{23}{24}(\Delta - \delta)^2 - \frac{3}{4}(\Delta - \delta),$$

and we conjecture that $L_r(\delta, r) = O((\Delta - \delta)^{r+1})$. Note that for $r > \Delta$, it is easy to see that $L_r(\delta, r) = 2^{\Delta-\delta} - 1$, as only the first loop is executed and Algorithm 3 degenerates to Algorithm 2.

Setting $\delta = 0$, we conjecture that the worst-case latency for processing a rank query according to Algorithm 3 is $O(\Delta^r)$, which is $O(\log^r n)$. The experimental results on the latency of RIPPLE in various queries and settings verify our conjecture.

4. TOP-K QUERIES

We first demonstrate the RIPPLE framework on top-k queries. Given a parameter k and a unimodal scoring function f, the top-k query retrieves a set of tuples A such that $|A| = k$ and $\forall \mathbf{t} \in A, \forall \mathbf{t}' \notin A : f(\mathbf{t}) \geq f(\mathbf{t}')$. A multivariate function f is unimodal if it has a unique local maximum.

In top-k processing, the abstract query Q comprises the scoring function f and the parameter k. The abstract state S is defined as m, τ, which indicates that m tuples with score above τ have already been retrieved.

With reference to the algorithms presented in Section 3, we next describe how the abstract functions of RIPPLE are materialized for top-k queries. The first function is computeLocalState, shown in Algorithm 4, which is used to construct an updated local state, given a forwarded global state. The function, executed on peer w, takes as input the query (f, k) and the global state (m^G, τ^G) and returns the local state (m_w^L, τ_w^L).

The main idea of computeLocalState is to identify as many high scoring local tuples as necessary to reach the goal of (globally) obtaining k tuples. Therefore, initially, peer w retrieves and stores in A up to k local tuples with score higher than τ^G (line 1). If the number of retrieved tuples plus those in the global state

received is less than k (line 2), peer w additionally retrieves tuples with lower than τ^G score (line 3). Upon completion of the computeLocalState algorithm, the local state m_w^L, τ_w^L is set to the number of local tuples retrieved and the lowest score among them, respectively.

Algorithm 4 w.top-computeLocalState(f, k, m^G, τ^G)

1: insert in A up to k local tuples with score better than τ^G
2: **if** $m^G + |A| < k$ **then**
3: insert in A up to $k - m^G - |A|$ highest ranking local tuples
4: **end if**
5: **return** $(m_w^L, \tau_w^L) \leftarrow (|A|, f(A))$

The computeGlobalState function, shown in Algorithm 5, derives the global state at w taking into account the forwarded global state (m^G, τ^G) and the current local state at w (m_w^L, τ_w^L). It just aggregates the number of tuples, and sets as threshold the lowest of the two thresholds.

Algorithm 5 w.top-computeGlobalState($f, k, m^G, \tau^G, m_w^L, \tau_w^L$)

1: **return** $(m_w^G, \tau_w^G) \leftarrow (m^G + m_w^L, \min\{\tau^G, \tau_w^L\})$

The computeLocalAnswer function, shown in Algorithm 6, extracts the local qualifying tuples using the local state. In the case of top-k processing, this means that all local tuples with score higher than the local threshold are retrieved.

Algorithm 6 w.top-computeLocalAnswer(f, k, m_w^L, τ_w^L)

1: insert in A all local tuples with score better than τ_w^L
2: **return** A

The next function we consider is updateLocalState, shown in Algorithm 7, which updates a local state given a set of local states. The function executed on peer w, takes as input the query (f, k) and a set of local states $(\{m_i^L, \tau_i^L\})$, and returns the local updated state (m_w^L, τ_w^L). Intuitively, updateLocalState attempts to find the highest possible threshold τ which guarantees the existence of k tuples.

Algorithm 7 w.top-updateLocalState($f, k, \{m_i^L, \tau_i^L\}$)

1: sort $\{m_i^L, \tau_i^L\}$ entries descending in their τ_i^L values
2: $m_w^L \leftarrow 0$
3: **for each** entry (m_i^L, τ_i^L) **do**
4: $m_w^L \leftarrow m_w^L + m_i^L$
5: $\tau_w^L \leftarrow \tau_i^L$
6: **if** $m_w^L \geq k$ **then break**
7: **end for**
8: **return** (m_w^L, τ_w^L)

Initially, w sorts the states descending based on their threshold values (line 1), and initializes the count of its local state counter m_w^L to zero (line 2). Then, it considers each local state in turn (lines 3–7), incrementing m_w^L (line 4) and setting the threshold to the currently considered local state's threshold (line 5). The examination of the states ends either when all local states have been considered, or when the number of tuples m_w^L reaches k (line 6).

Algorithm 8 decides if the region of a particular link of w contains qualifying tuples given the global state. A link should be considered if the number of tuples globally retrieved (to the best of w's knowledge) is less than k or if the region associated with the link has better ranked tuples than those globally retrieved. For the last check we use function f^+, which returns an upper bound on the score of any tuple within the given region.

Finally, Algorithm 9 compares two links based on how promising tuples their regions might contain. For this purpose, function f^+ is again used.

Algorithm 8 w.top-isLinkRelevant(i, f, k, m_w^G, τ_w^G)

1: **return** $m_w^G < k$ or $f^+(w.\text{link}[i].\text{region}) \geq \tau_w^G$

Algorithm 9 w.top-comp(i, j, f, k)

1: **return** $f^+(w.\text{link}[i].\text{region}) > f^+(w.\text{link}[j].\text{region})$

5. SKYLINE QUERIES

First, in Section 5.1, we discuss the instantiation of the RIPPLE framework for distributed processing of skyline queries. Then, in Section 5.2 we consider the case of the MIDAS overlay and propose an optimization.

5.1 Retrieving the Skyline

We describe distributed skyline query processing according to RIPPLE. We say that a tuple $\mathbf{t}$ dominates another $\mathbf{t}'$, denoted as $\mathbf{t} \succ \mathbf{t}'$, if $\mathbf{t}$ has better or as good values on all dimensions and strictly better on at least one dimension. Without loss of generality, we assume that in each dimension lower values are better. The skyline query retrieves all tuples that are not dominated by any other. In skyline query processing, the abstract query Q is empty. The abstract state S is defined as a set of not dominated tuples (partial skyline).

We first describe the computeLocalState method, depicted in Algorithm 10. Initially, peer w retrieves its local skyline, which serves as the local state (line 1). Then, it merges the tuples in the received global state and the local skyline, discarding the dominated ones, to construct the global state at w S_w^G (line 2). The final local state is computed as the intersection of the local skyline and the global state (line 3).

Algorithm 10 w.sky-computeLocalState(S^G)

1: $S_w^L \leftarrow$ the local skyline
2: $S_w^G \leftarrow$ computeSkyline($S^G \cup S_w^L$)
3: **return** $S_w^L \leftarrow S_w^L \cap S_w^G$

The computeGlobalState method, shown in Algorithm 11, sets the global state at w. As described before, S_w^G is the skyline computed over the received global state and the local skyline. Moreover, computeLocalAnswer, shown in Algorithm 12, returns the local tuples among those in the local state S_w^L.

Algorithm 11 w.sky-computeGlobalState(S^G, S_w^L)

1: **return** $S_w^G \leftarrow$ computeSkyline($S^G \cup S_w^L$)

The updateLocalState method, depicted in Algorithm 13 takes as input a set of local states $\{S_i^L\}$ and combines them to produce an updated local state. In particular, peer w merges all local states and computes their skyline, which becomes the updated local state at w.

Then, we detail the isLinkRelevant method. Algorithm 14 iterates the tuples in the global state (lines 1–5). If any of them dominates the entire region of the link (i.e., it dominates any possible tuple within the region), then this link certainly contains no skyline tuple, and the method returns false (line 3). Otherwise the link's region should be considered (line 6).

Finally, the comp function, shown in Algorithm 15, compares the regions of two links. The link whose region is closer to the origin of the axes $\mathbf{0}$ is better. Note that function d^- computes the minimum distance of any tuple in a region from $\mathbf{0}$.

5.2 An Optimization for MIDAS

In this section, we present an optimization for improving the efficiency of the distributed skyline computation when RIPPLE is used on top of the MIDAS DHT. The main intuition behind this

Algorithm 12 w.sky-computeLocalAnswer(S_w^L)

1: **return** $A \leftarrow$ local tuples of S_w^L

Algorithm 13 w.sky-updateLocalState($\{S_i^L\}$)

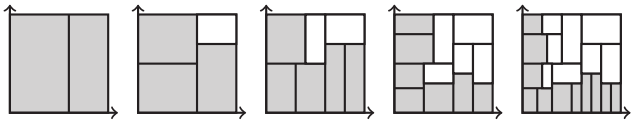
1: **return** $S_w^L \leftarrow$ computeSkyline($\bigcup_i S_i^L$)

approach is that we want the peer that receives a request to be part of the skyline more often than not. Therefore, message overhead would be reduced if we could target requests towards peers that are located as close as possible to the borders of the keyspace, since it may contain not dominated tuples.

To understand this, observe that if the RIPPLE algorithm run in a peer located in the middle of the domain, it would return no or insignificant tuples consisting of false positives (i.e., they most probably will be dominated by tuples of another peer). On the downside, not necessarily all nodes located by the borderlines contain tuples belonging to the skyline, even though some definitely contain.

So, the question is how to locate the peers at the boundaries. Recall from Section 2.3 that the MIDAS overlay resembles a virtual distributed k-d tree. Each peer w has links to peers that reside within its sibling subtrees. Note that MIDAS does not specify *which specific* peer w should have as its neighbor. Therefore, there is some freedom in the structure of MIDAS, which we try to take advantage of for the benefit of the distributed skyline computation.

MIDAS allows us to identify the identifiers of peers that are positioned on the lower borders of the domain. Figure 2 illustrates the two- dimensional case where the dimension that the split takes place changes at each level. The peers whose identifiers satisfy either one of the regular expressions $p_h = (X0)^*X?$ and $p_v = (0X)^*0?$ are shaded, where X denotes either 1 or 0 ($X \leftarrow (0|1)$). Note that the peers that have a 0 at every other digit are responsible for the lower parts of the domain along the horizontal and the vertical bounds, respectively. Likewise, for D dimensions where the split dimension alternates sequentially, we have D patterns in total with $p_0 = (X0\cdots0)^*X0\{0, D-1\}$, $p_1 = (0X0\cdots0)^*0X0\{0, D-2\}$, $p_2 = (00X0\cdots0)^*00X0\{0, D-3\}$, and so on. It is not hard to show that if a peer has an id that does not accord with any of the patterns, then naturally, none of its derived peers will, regardless of the number and type of splits. This is due to the fact that its id will be the prefix of all its derived peers.



(a) level 1 (b) level 2 (c) level 3 (d) level 4 (e) level 5

Figure 2: Overlay nodes with identifiers of the form $p_h = (X0)^*X?$ and $p_v = (0X)^*0?$ for the two-dimensional case.

Now, we will force the links of a peer to have an identifier according to some pattern $p_0, \cdots, p_{D-1}$, if possible. This recursive procedure is now incorporated in the join protocol and is run as the

Algorithm 14 w.sky-isLinkRelevant(i, S_w^G)

1: **for each** $\mathbf{s} \in S_w^G$ **do**
2: **if** $\mathbf{s} \succ w$.link[i].$region$ **then**
3: **return false**
4: **end if**
5: **end for**
6: **return true**

Algorithm 15 w.sky-comp(i, j)

1: **return** $d^-(w.\text{link}[i].\text{region}, \mathbf{0}) < d^-(w.\text{link}[j].\text{region}, \mathbf{0})$

overlay inflates. This means, in practice, that the j-th link of any peer is established so as to have an identifier that complies with one of the aforementioned patterns, if there is at least one such peer within the sibling subtree at level j. The original MIDAS policy suggests examining only the j first bits of the links' identifiers. We now impose the following policy when forming the structure of the overlay as new peers join. When a new peer joins and an existing peer splits its zone into two. Then the two peers, the new and the one who split its zone, become siblings. The following procedure takes place.

1. No one or both peers are associated with ids that obey the pattern. Then, the peers that were linked to the original peers are now associated with any of the two new peers.
2. Only one of the two peers has an id that obeys the pattern. Then, all back-links of the original peer are now assigned to the peer that satisfies the pattern. Now, only its sibling is directly connected to the peer with the indifferent pattern.

(a) start (b) 1st hop (c) 2nd hop (d) 3rd hop (e) 4th hop

Figure 3: Processing skyline queries with RIPPLE's fast algorithm for the two-dimensional case.

Figures 3 illustrate the effect of the MIDAS structural optimization, when processing skyline queries based on RIPPLE's fast extreme case. With gray color are drawn the peers whose ids obey the two patterns. In each hop, light green indicates which peer is processing the query, while dark green indicates the peers that have processed the query so far. Notice how RIPPLE over MIDAS efficiently targets the gray peers, which potentially contain answer tuples.

6. DIVERSIFICATION

Section 6.1 establishes the necessary background. Section 6.2 introduces a RIPPLE-based algorithm for an important sub-problem. Then Section 6.3 details our solution to the diversification problem.

6.1 Preliminaries

Given a query point $\mathbf{q}$, the *k-diversification query* is to find a set O_k of k tuples that maximizes the following objective function:

$$f(O, \mathbf{q}) = \lambda \max_{\mathbf{x} \in O} d_r(\mathbf{x}, \mathbf{q}) - (1-\lambda) \min_{\mathbf{y}, \mathbf{z} \in O} d_v(\mathbf{y}, \mathbf{z}). \quad (1)$$

The first part of the objective function is defined by the maximum distance of any tuple in O from the query q. A low value of this part indicates that the set O contains relevance tuples. The second part of the objective function is defined by the minimum distance between any two tuples in O. A high value of this part indicates that the set O contains diverse tuples. The distances in Equation 1 are computed by user-defined functions d_r, d_v. The λ user parameter takes value in $[0, 1]$ and controls the relative weights of relevance and diversity. Overall, the goal of the k-diversification query is to find a set O_k which strikes the desirable balance between the relevance and diversity of its tuples.

An important sub-problem, which is encountered in most algorithms that greedily solve the k-diversification query, is the following. Given a query point $\mathbf{q}$ and a set of objects O, the *single tuple diversification query* is to find a tuple $\mathbf{t}^* \notin O$ which minimizes the

objective function for the set $O \cup \{\mathbf{t}^*\}$, i.e.,

$$\mathbf{t}^* = \underset{\mathbf{t} \notin O}{\operatorname{argmin}} f(O \cup \{\mathbf{t}\}, \mathbf{q}). \quad (2)$$

Looking into the effect of adding a new tuple $\mathbf{t}$ into O, we discern four distinct cases. According to the first, tuple $\mathbf{t}$ is within range of the least relevant object in O, and also farther from any object in O than the distance between the closest pair of tuples in O. Therefore, the value of f for the augmented set does not change as the least relevant tuple and the least distant pair in $O \cup \{\mathbf{t}\}$ remain the same.

Next, according to the second case, the new tuple $\mathbf{t}$ is farther from $\mathbf{q}$ than any object in O, and farther from any object in O than the distance between the closest pair of tuples in O. Therefore, the objective value of the augmented set is increased by the relevance difference between $\mathbf{p}$ and the former farthest tuple from $\mathbf{q}$.

The third case is when even though $\mathbf{t}$ is closer to $\mathbf{q}$ than the least relevant tuple in O, its closest distance from any tuple in O is less than the distance between the closest pair of tuples in O. Hence, the objective value of the augmented set increases by this difference.

Last, if $\mathbf{t}$ is less relevant than any tuple in O, and its distance from any tuple in O is less than the distance between any pair of tuples in O, then $f(O \cup \{\mathbf{t}\})$ increases by both the relevance and diversity loss caused by the inclusion of $\mathbf{t}$ in O.

Taking these observations into account, given a set of objects O and the query $\mathbf{q}$, we define a scoring function $\phi(\mathbf{t}, \mathbf{q}, O)$ for tuples, shown in Equation 3. The four cases discussed before, correspond to the four clauses of the objective score. It is easy to verify that the tuple which minimizes Equation 3 also solves the single tuple diversification query. Furthermore, note that it is possible to construct ϕ functions for objective functions other than Equation 1; we omit details in the interest of space.

6.2 Single Tuple Insertion

This section describes how to apply the RIPPLE framework to solve the single tuple diversification query. As before, we instantiate the abstract functions used in the ripple algorithm. For the specific problem at hand, the query Q contains the query point $\mathbf{q}$, the set of objects O, and the scoring function ϕ (derived from the objective function f). The state S corresponds to a threshold score τ. The answer A is the tuple $\mathbf{t}^* \notin O$ that minimizes the scoring function.

We first describe the computeLocalState function, shown in Algorithm 16, which derives the local state given a transmitted global state. Initially, peer w retrieves the local tuple $\mathbf{t}$ that minimizes function ϕ (line 2). Then if its score is less than the global state/threshold τ^G (line 2) the local state is initialized to its score (line 3). Otherwise, the local state becomes equal to the global state (line 5), meaning that no local tuple is better than one already found. In any case, function computeGlobalState, shown in Algorithm 17, sets the global state at peer w to the local state.

Algorithm 16 w.div-computeLocalState($\mathbf{q}, O, \phi, \tau^G$)

```
1: t ← w.getMostDiverseLocalObject(q, O, φ)
2: if φ(t, q, O) < τ^G then
3:     return τ_w^L ← φ(t, q, O)
4: else
5:     return τ_w^L ← τ^G
6: end if
```

Algorithm 17 w.div-computeGlobalState($\mathbf{q}, O, \phi, \tau^G, \tau_w^L$)

```
1: return τ_w^G ← τ_w^L
```

Function computeLocalAnswer, depicted in Algorithm 18, extracts the local tuple, which is currently the best answer if it exists. Initially, peer w retrieves the local tuple $\mathbf{t}$ that has the lowest score (line 1). Subsequently, if its score is equal to the local state (line 2), tuple $\mathbf{t}$ becomes the local answer (line 3). Otherwise, the local answer is empty (line 5).

Algorithm 18 w.div-computeLocalAnswer($\mathbf{q}, O, \phi, \tau_w^L$)

```
1: t ← w.getMostDiverseLocalObject(q, O, φ)
2: if φ(t, q, O) = τ_w^L then
3:     return A ← t
4: else
5:     return A ← null
6: end if
```

Updating the local state upon receiving a set of local states is shown in Algorithm 19. Peer w simply sets its local state to the minimum among those received. Algorithm 20 decides whether the region assigned to the i-th link of peer w is worth visiting. The decision is based on whether a lower bound on the score of any tuple in the region is lower than the global state. Function ϕ^- computes this lower bound. Finally, Algorithm 21 compares the priority of w's links. The one whose region has the lowest lower bound on score, given by ϕ^-, has the highest priority.

Algorithm 19 w.div-updateLocalState($\mathbf{q}, O, \phi, \{\tau_i^L\}$)

```
1: return τ_w^L ← min_i{τ_i^L}
```

Algorithm 20 w.div-isLinkRelevant($i, \mathbf{q}, O, \phi, \tau_w^G$)

```
1: return φ⁻(w.link[i].region, q, O) < τ_w^G
```

Algorithm 21 w.div-comp($i, j, \mathbf{q}, O, \phi$)

```
1: return φ⁻(w.link[i].region, q, O) < φ⁻(w.link[j].region, q, O)
```

6.3 Solving the Diversification Problem

Building upon the RIPPLE-based solution to the single tuple diversification query, described in the previous section, we propose a greedy algorithm for solving the k-diversification query.

Algorithm 22 shows the pseudocode of our solution, executed on the initiator peer v. Initially, a set of k tuples is retrieved from the network by invoking the initialize function (line 2). This function can be as simple as retrieving k random tuples, or more elaborate solving k times the single tuple diversification query, by invoking algorithm div-ripple, discussed in the previous section.

Then given an initial set O of k tuples, the algorithm attempts to improve on the objective value of the set by performing a series of iterations (lines 2–9). Each pass consists of a call to the div-improve algorithm, which we explain later, to obtain a new set of tuples (line 3). The iterations terminate prematurely if div-improve cannot construct a better set (line 7).

We now discuss the div-improve method, shown in Algorithm 23. Its goal is to determine a single tuple $\mathbf{t}_{out} \in O$ to replace with tuple $\mathbf{t}_{in} \notin O$, so that the objective value of O improves. Initially, these tuples are set to null (lines 1–2).

Then, div-improve obtains an ordering on the tuples of O (line 3). Each tuple $\mathbf{t}_i \in O$ is given a score computed by the ϕ function as $\phi(\mathbf{t}_i, q, O \setminus \{\mathbf{t}_i\})$, i.e., tuple $\mathbf{t}_i$ is excluded from O when computing its score. The ordering is descending on the tuples' scores. Observe that if we consider the sets $O \setminus \{\mathbf{t}_i\}$, the ordering implies that they are ordered ascending on their objective values. As a result, the first tuple has the worst score, but the set obtained by removing it has the best objective value. To understand this, assume tuple $\mathbf{t}_i$ is ordered before $\mathbf{t}_j$, i.e., $\phi(\mathbf{t}_i, \mathbf{q}, O \setminus \{\mathbf{t_i}\}) \geq$

$$\phi(\mathbf{t},\mathbf{q},O)=\begin{cases}0,\ \text{if } d_r(\mathbf{t},\mathbf{q})\leq \max_{\mathbf{x}\in O} d_r(\mathbf{x},\mathbf{q}) \text{ and } \min_{\mathbf{x}\in O} d_v(\mathbf{t},\mathbf{x})\geq \min_{\mathbf{y},\mathbf{z}\in O} d_v(\mathbf{y},\mathbf{z}),\\ \lambda(d_r(\mathbf{t},\mathbf{q})-\max_{\mathbf{x}\in O} d_r(\mathbf{x},\mathbf{q})),\ \text{if } d_r(\mathbf{t},\mathbf{q})> \max_{\mathbf{x}\in O} d_r(\mathbf{x},\mathbf{q}) \text{ and } \min_{\mathbf{x}\in O} d_v(\mathbf{t},\mathbf{x})\geq \min_{\mathbf{y},\mathbf{z}\in O} d_v(\mathbf{y},\mathbf{z}),\\ (1-\lambda)(\min_{\mathbf{x},\mathbf{y}\in O} d_v(\mathbf{x},\mathbf{y})-\min_{\mathbf{z}\in O} d_v(\mathbf{t},\mathbf{z})),\ \text{if } d_r(\mathbf{t},\mathbf{q})\leq \max_{\mathbf{x}\in O} d_r(\mathbf{x},\mathbf{q}) \text{ and } \min_{\mathbf{x}\in O} d_v(\mathbf{t},\mathbf{x})< \min_{\mathbf{y},\mathbf{z}\in O} d_v(\mathbf{y},\mathbf{z}),\\ \lambda(d_r(\mathbf{t},\mathbf{q})-\max_{\mathbf{x}\in O} d_r(\mathbf{x},\mathbf{q}))+(1-\lambda)(\min_{\mathbf{y},\mathbf{z}\in O} d_v(\mathbf{y},\mathbf{z})-\min_{\mathbf{x}\in O} d_v(\mathbf{t},\mathbf{x})),\ \text{otherwise.}\end{cases} \tag{3}$$

Algorithm 22 v.diversify($\mathbf{q}, k$)

```
1: O ← v.initialize(q, k)
2: for i ← 1 to MAX_ITERS do
3:     O' ← v.div-improve(q, O)
4:     if O' ≠ O then
5:         O ← O'
6:     else
7:         break
8:     end if
9: end for
```

Algorithm 23 v.div-improve($\mathbf{q}, O$)

```
1: t_in ← null
2: t_out ← null
3: sort tuples in O descending on their φ scores
4: for each t_i ∈ O do
5:     if t_in = null then
6:         τ ← φ(t_i, q, O)
7:     else
8:         τ ← f(O \ {t_out} ∪ {t_in}, q) − f(O, q)
9:     end if
10:    v.div-ripple(v, v, q, O \ t_i, τ, R, r)
11:    t_in ← v.receive()
12:    if t_in ≠ null then
13:        t_out ← t_i
14:    end if
15: end for
16: return O \ {t_out} ∪ {t_in}
```

$\phi(\mathbf{t}_j,\mathbf{q},O\setminus\{\mathbf{t_j}\})$, and consider the following equation:

$$\begin{aligned} f(O,\mathbf{q}) = f(O,\mathbf{q}) \quad &\Leftrightarrow\\ f(O\setminus\{\mathbf{t_i}\}\cup\{\mathbf{t_i}\},\mathbf{q}) = f(O\setminus\{\mathbf{t_j}\}\cup\{\mathbf{t_j}\},\mathbf{q}) \quad &\Leftrightarrow\\ f(O\setminus\{\mathbf{t_i}\},\mathbf{q})+\phi(\mathbf{t}_i,\mathbf{q},O\setminus\{\mathbf{t_i}\}) = &\\ f(O\setminus\{\mathbf{t_j}\},\mathbf{q})+\phi(\mathbf{t}_j,\mathbf{q},O\setminus\{\mathbf{t_j}\}) \quad &\Leftrightarrow\\ f(O\setminus\{\mathbf{t_i}\},\mathbf{q})\leq f(O\setminus\{\mathbf{t_j}\},\mathbf{q}).& \end{aligned}$$

Therefore, $O\setminus\{\mathbf{t_i}\}$ has better objective value. Overall, the rationale is that by considering good sets first, it becomes more likely to find a good replacement early.

Algorithm div-improve examines each tuple in turn (lines 4–15). Briefly, each turn considers the case of removing the tuple under examination from O, and searches for the best tuple outside O to include. The algorithm requires this replacement to result in a set with better objective value than that of the original set and any previously considered set.

To find the best replacement tuple when tuple $\mathbf{t}_i$ is considered, div-improve invokes the div-ripple algorithm of the previous section, using the set $O\setminus\{\mathbf{t}_i\}$ as input (line 10). Contrary to a regular initial invokation of div-ripple, the initiator includes a global state τ in its call. Note that regularly the initial global state would be set to a neutral value like ∞. However, in this case we explicitly set τ to enforce the requirement that the replacement tuple should result in a set with better objective value.

When no suitable replacement tuple is found yet (line 5), the global state is set to the ϕ score of tuple $\mathbf{t}_i$ (line 6). This makes div-ripple search for a tuple which when added would result in set with better objective value than the original set. If the algorithm has already found a tuple $\mathbf{t}_{out}$ to replace with $\mathbf{t}_{in}$, the global state is set to the objective value of this improved set minus the objective value of the original set (line 8). The intuition is to look for a tuple which can improve the objective value even more.

Initializing a global state in this manner, expedite the search as it prunes large parts of the space. As a result, no replacement tuple may be found. Otherwise, the current best tuple to insert and remove are set (lines 11, 13, respectively). At the end of the algorithm the improved set is returned.

Table 1: Experimental Configuration

Parameter	Range	Default
overlay size	$2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16}, 2^{17}$	2^{14}
dimensions	$2, 3, 4, 5, 6, 7, 8, 9, 10$	$5, 6$
result-size	10,20,30,40,50,60,70,80,90,100	10
rel/div tradeoff	$0, 0.2, 0.3, 0.5, 0.7, 0.8, 1$	0.5

7. EXPERIMENTAL EVALUATION

To assess our methods and validate our analytical claims, we simulate a dynamic network environment and study query performance.

7.1 Setting

Methods. In order to evaluate the performance of our framework in different queries, we implemented various methods from the literature. Note that RIPPLE is showcased over the MIDAS index. Regarding skyline queries, we implement DSL [20], which relies on CAN [13], and SSP [18], which exploits a Z-curve over BATON [10]. For k-diversification queries, we adapt the algorithm of [12], termed baseline, for a distributed setting based on CAN. For fairness, we force both heuristic diversification algorithms to produce the same result at each step. Hence our metrics capture directly the cost/performance of methods and are not affected by the quality of the result.

Overlay. We simulate a dynamic topology that captures arbitrary physical peer joins and departures, in two distinct stages. In the *increasing stage*, physical peers continuously join the network while no physical peer departs. It starts from a network of 1,024 physical peers and ends at 131,072 physical peers. On the other hand, in the *decreasing stage*, physical peers continuously leave the network while no new physical peer joins. This stage starts from a network of 131,072 physical peers and ends when only 1,024 physical peers are left. When we vary the network size, the figures show the results during the increasing stage; the results during the decreasing stage are analogous and omitted.

Parameters. Our experimental evaluation examines four parameters. The network size is varied from 1,024 up to 131,072 physical peers. The number of dimensions considered varies from 2 up to 10. We also investigate the effect of the result-size k in top-k and diversification queries, i.e., the number of expected items in a result, varying it from 10 up to 100. For diversification, we also study the trade-off between relevance and diversity by tweaking the weight λ in Equation 3 from 0 up to 1. The tested ranges and default values for these parameters are summarized in Table 1. When we vary one parameter, all others are set at their default values.

Metrics. Regarding query processing performance, we employ two main metrics. First, *latency* measures the number of hops required during processing, where lower values suggest faster response. Moreover, distributed query processing imposes a load on multiple physical peers, including ones that may not contribute to the answer. Therefore, we study another metric. Specifically, *congestion* is defined as the average number of queries processed at any peer when n uniformly queries are issued (n is the network size), as lower values suggest lower load. This actually resembles the average traffic a peer intakes when n queries are issued.

Data and Queries. In top-k and skyline queries, we use a dataset, denoted as NBA, consisting of 22,000 six-dimensional tuples with NBA players statistics* covering seasons from 1946 until 2009. In particular, we used the points, rebounds, assists and blocks per game attributes. A top-k query on this dataset retrieves the best all-around players, as individual statistics are aggregated by the scoring function. A skyline query on this dataset retrieves the players who excel in particular or combinations of statistics.

In k-diversification queries, we use a collection, denoted as MIRFLICKR, of 1,000,000 images widely used in the evaluation of content-based image retrieval methods†. We extracted the five-bucket edge histogram descriptors, of the MPEG-7 specification, as the feature vector. The L_1 distance norm is used for the relevance and diversity scores.

In order to study the impact of dimensionality on all types of queries we construct clustered, synthetic, multi-dimensional datasets in $[0, 1]^D$, denoted as SYNTH. Specifically, they consist of 1,000,000 records of varied dimensionality from 2 up to 10, generated around 50,000 cluster centers according to a zipfian distribution with skewness factor equal to $\sigma = 0.1$.

Note that every reported value in the figures is the average of executing 65,536 queries over 16 distinct networks.

7.2 Experimental Results

7.2.1 Top-k Queries

Since there is no competitor method for top-k queries, this section serves as a benchmark for the effect of the ripple parameter r. In particular, we consider four r values: the two extreme values, 0, where RIPPLE executes the fast algorithm, and Δ, where RIPPLE corresponds to slow, and two intermediate values, $\Delta/3$, $2\Delta/3$.

In our experiments, we use the NBA dataset in Figures 4 and 6, and SYNTH in Figure 5. As expected, low r-values (close to 0) are translated into fast responsiveness, though, at a relatively higher communication cost, whilst at high r-values (close to the maximum number of neighbors Δ) message overhead is minimized as only highly relevant peers are burdened.

Figure 4(a) shows that latency scales very well as the overlay grows. Even for high r values and the extreme setting of Δ, due to prioritization in RIPPLE, latency is much lower than the worst-case linear cost and scales polylogarithmically. Conversely, the increased congestion for low r values, shown in Figure 4(b), is explained by the parallel transmission to the neighbors of each encountered peer.

Dimensionality affects performance only slightly, as shown in Figure 5. The reason is that the core structure (number of neighbors per peer, overlay size) of the underlying index (MIDAS) determining performance is not affected; only the dimensionality of the zones changes. Finally, Figure 6 shows that increasing the requested number or results has negative effect on both latency and congestion, as the total number of accessed and relevant peer increases. In particular, note that the value of $k = 100$ is quite high corresponding to approximately 0.5% of the NBA dataset size.

*Available at `http://www.basketball-reference.com`

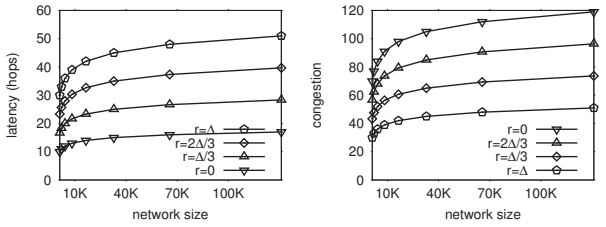
†Available at `http://press.liacs.nl/mirflickr/`

7.2.2 Skyline Queries

For the remainder of the experimental evaluation, we only consider the extreme values for the ripple parameter. In particular, we denote as ripple-fast the case of $r = 0$, and as riple-slow the case of $r = \Delta$. The latency and congestion for other values of r lies in between the two extremes, as demonstrated in the previous section.

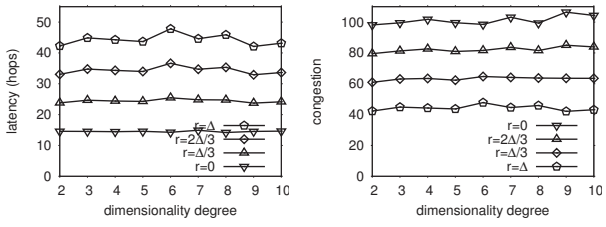
The evaluation of RIPPLE on skyline queries is shown in Figures 7 and 8 using the NBA and SYNTH datasets, respectively. In Figure 7(a) latency shows a logarithmic behavior for ripple-slow and SSP, due to the exploited properties of their indexing infrastructures. Nevertheless, SSP is not as efficient because it does not rely on a pure multi-dimensional index, unlike MIDAS, and maps multi-dimensional keys to a unidimensional space-filling curve instead. Therefore, more false positive skyline tuples are considered and network routing becomes less effective with increased dimensionality, taking its toll on latency and message overhead.

In Figure 7(a), DSL appears to be slower as messages are forwarded strictly to adjacent peers whose zone abuts in all but one dimension. Nevertheless, DSL is in position of exploiting the increased number of dimensions in Figure 8. In particular, the diameter of the overlay decreases dramatically as each peer has significantly more neighbors due to the increased number of established links. In other words, larger neighborhoods is translated in practice into better and more efficient routing, as queries are forwarded to more highly relevant peers, selected from a wider range of links as dimensionality increases. However, this comes at an increased



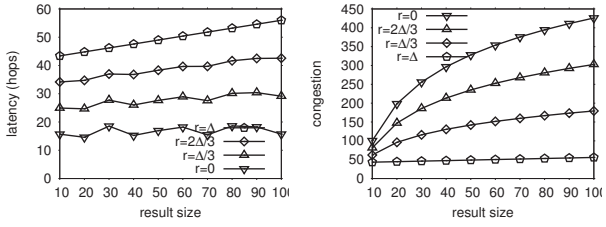
(a) latency (b) congestion

Figure 4: Top-k query performance in terms of overlay size.



(a) latency (b) congestion

Figure 5: Top-k query performance in terms of dimensionality.



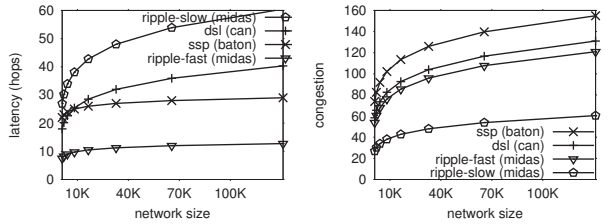
(a) latency (b) congestion

Figure 6: Top-k query performance in terms of result size.

maintainance cost for DSL, which increases linearly with dimensionality. This cost corresponds to network information, maintenance and links each peer has to preserve up-to-date. In any case, this method is clearly inept for low dimensionality datasets as both latency and congestion deteriorate in Figure 8.

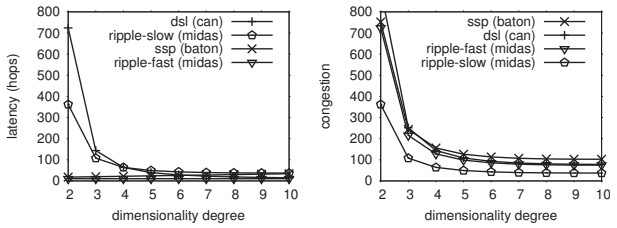
Alhough the slowest, ripple-slow consumes the least resources in Figures 7(b) and 8(b). However, it does not perform well for low dimensionality spaces in terms of latency due to the sequential access of peers and the large number of relevant peers (network size is fixed in Figure 8). Nevertheless, it performs better than what the worst case analysis predicts. In practice, due to prioritizing the peers that process the query according to their possibility of participating in the skyline set, we expect queries to resolve much faster than in linear time.

In general, congestion appears to be relatively high for all methods, in a sense that these operations appear to be expensive, but this is only due to the large number of relevant peers. For instance, approximately $d\sqrt[d]{n} + m$ peers are relevant, and hence, even more will have to be accessed, where n stands for the overlay size, d for the dimensionality degree of the problem, and m the number of encountered peers that are not located by the borderlines of the keyspace, either due to false positives, e.g. during the early steps of the algorithm, or because they were not dominated by any other peer accessed at the time. Therewith, the main challenge in distributed skyline processing is how should all these peers be accessed, and more importantly at what cost, in terms of latency, congestion, message overheads. In essence, we propose a toolset for skyline computation with tunable performance, ranging from ripple-fast, which is very fast, up to ripple-slow which although slower, consumes very little network resources.



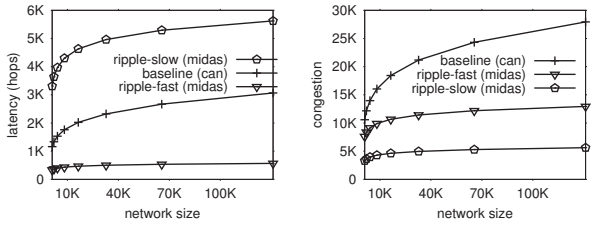
(a) latency (b) congestion

Figure 7: Skyline computation in terms of overlay size.



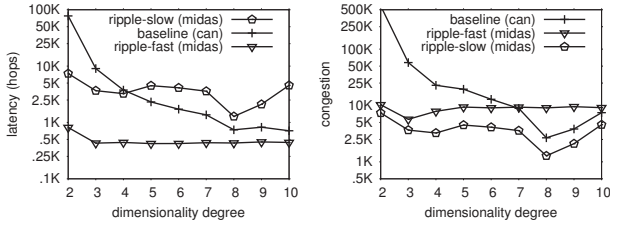
(a) latency (b) congestion

Figure 8: Skyline computation in terms of dimensionality.



(a) latency (b) congestion

Figure 9: Diversification performance in terms of overlay size.



(a) latency (b) congestion

Figure 10: Diversification performance in terms of dimensions.

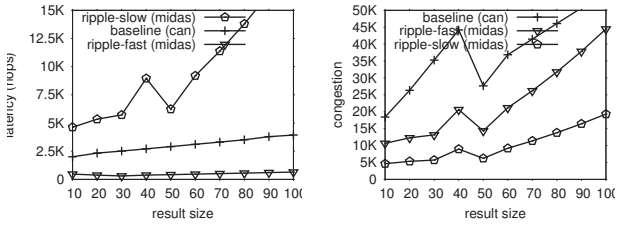
7.2.3 k-Diversification Queries

We next compare our RIPPLE-based diversification algorithm to the baseline method, in terms of network latency and congestion. As before, we consider the extreme cases of our framework, labelled ripple-slow and ripple-fast. Figure 9 presents results on the MIRFLICKR dataset with respect to the overlay size, Figure 10 shows results on SYNTH while varying the dimensionality, Figure 11 varies the result size for the MIRFLICKR dataset, and Figure 12 studies the relative weight λ using the MIRFLICKR dataset.

Apparently, ripple-fast is much faster than baseline for any number of overlay peers and dimensions, as shown in Figures 9(a) and 10(a). Additionally, the benefits of RIPPLE become evident in Figure 9(b) where network congestion for our paradigm diminishes substantially.

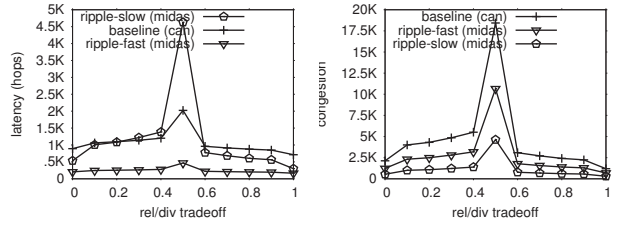
Moreover, the required number of iterations for the RIPPLE-based diversification algorithm to converge plays a prevalent role in the performance of the methods. Nevertheless, we note that performance is affected by both the effectiveness of the diversified search methods and the indexing infrastructure used. This is evident in Figure 10 where the baseline's performance ameliorates with dimensionality, as the number of links established in each peer increases analogously and routing becomes more effective.

Also note that the number of relevant peers with each iteration diminishes for the RIPPLE-based methods. In Figures 9(b) and 10(b), which illustrate network congestion, limiting our search only to the regions that contain tuples with improved scores with ripple-



(a) latency (b) congestion

Figure 11: Diversification performance in terms of result size.



(a) latency (b) congestion

Figure 12: Diversification performance for rel/div tradeoff.

slow is significantly better. Specifically, RIPPLE requires only a small portion of the messages the competitor needs.

Figure 11 exhibits the impact of the cardinality of the answer-set on performance. Apparently, the increase of the result-size has a bilateral impact on performance. Specifically, as more items need to be examined in the result-set and whether they should be replaced, we would expect a linear increase in latency and congestion with k that would impair performance due to the additional consecutive operations for computing all possible replacements. However, this is not the case for ripple-fast in Figure 11(a), where the combined impact of two contradicting phenomena is revealed. To elaborate, since we examine only one item from the result at a time, there are $k-1$ other items restricting the searched area of the domain. We note, that only the overlay peers that overlap with the intersection of all $k-1$ restricted search areas are accessed. Therefore, as k increases, there are more restrictions imposed which effectively make the search area shrink, and therefore, less peers are encountered. As a result, performance seems to be unaffected for ripple-fast in the more selective search operations. However, these beneficial effects cease to help when k takes very high values, and hence, the processing cost was the dominant performance factor. Which of the two phenomena will prevail each time depends on the effectiveness of our pruning policy. Besides, congestion increases slowly with k for our methods in Figure 11(b) for the same reasons.

Figure 12 shows an interesting pattern. When λ takes very low or very high values the number of encountered overlay peers diminishes dramatically, and therewith, the number of hops required to access them. In substance, diversified search becomes very limited, as the proper areas of the domain that contain highly ranked items are either close to the query tuple for $\lambda \to 1$, where very relevant items are promoted (enclosed search area around the query point), or are located along the borders of the domain for $\lambda \to 0$, as tuples that are distant to each other are promoted mostly. Therefore, when λ takes values close to 0 or 1, the performed search is pretty much automatically directed towards these areas. In other words, diversified search qualifies small parts of the domain for either very low or high values of λ, and thereby, query processing is limited to certain overlay peers responsible for these specific areas. This effect is illustrated in Figure 12, where both response time and bandwidth consumption decrease as we move further away from $\lambda = 0.5$.

8. CONCLUSIONS

This work has addressed the problems of efficient distributed processing of top-k, skyline, and k-diversification queries, in the context of large-scale decentralized networks, by introducing a unified framework, called RIPPLE. Our methods investigate the trade-off between optimal latency and congestion through a single parameter. The key ideas of RIPPLE is to take advantage of local information regarding query processing so as to better guide the search. The instantiation of our framework for skyline queries has resulted in an efficient distributed algorithm, while for the case of diversification queries, it constitutes the first work on the subject.

9. ACKNOWLEDGMENTS

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

10. REFERENCES

[1] R. Akbarinia, E. Pacitti, and P. Valduriez. Reducing network traffic in unstructured p2p systems using top-*k* queries. *Distributed and Parallel Databases*, 19(2-3):67–86, 2006.

[2] W.-T. Balke, W. Nejdl, W. Siberski, and U. Thaden. Progressive distributed top k retrieval in peer-to-peer networks. In *ICDE*, 2005.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.

[4] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *PODC*, 2004.

[5] J. G. Carbonell and J. Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *SIGIR*, 1998.

[6] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[7] S. Gollapudi and A. Sharma. An axiomatic framework for result diversification. *IEEE Da. Eng. Bul.*, 32(4):7–14, 2009.

[8] K. Hose and A. Vlachou. A survey of skyline processing in highly distributed environments. *VLDB J.*, 21(3).

[9] 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.

[10] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: A balanced tree structure for peer-to-peer networks. In *VLDB*, 2005.

[11] S. Michel, P. Triantafillou, and G. Weikum. Klee: A framework for distributed top-k query algorithms. In *VLDB*, 2005.

[12] E. Minack, W. Siberski, and W. Nejdl. Incremental diversification for very large sets: a streaming-based approach. In *SIGIR*, 2011.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM*, 2001.

[14] N. H. Ryeng, A. Vlachou, C. Doulkeridis, and K. Nørvåg. Efficient distributed top-*k* query processing with caching. In *DASFAA*, 2011.

[15] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[16] G. Tsatsanifos, D. Sacharidis, and T. Sellis. Index-based query processing on distributed multidimensional data. *GeoInformatica*, 17(3):489–519, 2013.

[17] A. Vlachou, C. Doulkeridis, K. Nørvåg, and M. Vazirgiannis. On efficient top-k query processing in highly distributed environments. In *SIGMOD*, 2008.

[18] S. Wang, B. C. Ooi, A. K. H. Tung, and L. Xu. Efficient skyline query processing on peer-to-peer networks. In *ICDE*, 2007.

[19] S. Wang, Q. H. Vu, B. C. Ooi, A. K. H. Tung, and L. Xu. Skyframe: a framework for skyline query processing in peer-to-peer systems. *VLDB J.*, 18(1):345–362, 2009.

[20] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. El Abbadi. Parallelizing skyline queries for scalable distribution. In *EDBT*, 2006.

[21] K. Zhao, Y. Tao, and S. Zhou. Efficient top-k processing in large-scaled distributed environments. *Data Knowl. Eng.*, 63(2):315–335, 2007.

Efficient Influence-Based Processing of Market Research Queries

Anastasios Arvanitis*
National Technical University of Athens, Greece
anarv@dblab.ntua.gr

Antonios Deligiannakis
Technical University of Crete, Greece
adeli@softnet.tuc.gr

Yannis Vassiliou
National Technical University of Athens, Greece
yv@cs.ntua.gr

ABSTRACT

The rapid growth of social web has contributed vast amounts of user preference data. Analyzing this data and its relationships with products could have several practical applications, such as personalized advertising, market segmentation, product feature promotion etc. In this work we develop novel algorithms for efficiently processing two important classes of queries involving user preferences, i.e. potential customers identification and product positioning. With regards to the first problem, we formulate product attractiveness based on the notion of reverse skyline queries. We then present a new algorithm, termed as RSA, that significantly reduces the I/O cost, as well as the computation cost, when compared to the state-of-the-art reverse skyline algorithm, while at the same time being able to quickly report the first results. Several real-world applications require processing of a large number of queries, in order to identify the product characteristics that maximize the number of potential customers. Motivated by this problem, we also develop a batched extension of our RSA algorithm that significantly improves upon processing multiple queries individually, by grouping contiguous candidates, exploiting I/O commonalities and enabling shared processing. Our experimental study using both real and synthetic data sets demonstrates the superiority of our proposed algorithms for the studied classes of queries.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*

Keywords

reverse skylines, preferences, market research

1. INTRODUCTION

Analyzing user data (e.g., query logs, purchases) has seen considerable attention due to its importance in providing insights regarding users' intentions and helping enterprises in the process of decision making. Recently, the rapidly growing social web has been a source of vast amounts of data concerning user preferences in the form of ratings, shares, likes, etc. Previous efforts (e.g., in preference learning and recommender systems) mainly focus on helping users discover the most interesting, according to their preferences, among a pool of available products.

Highlighting the manufacturer's perspective, the need for tools to analyze user preferences for improving business decisions has been well recognized. Preference analysis has various important applications such as personalized advertising, market segmentation, product positioning etc. For example, a laptop manufacturer might be interested in finding those users that would be more interested in purchasing a laptop model. Thereby, manufacturers can benefit by targeting their advertising strategy to those users. Or they might search for laptop feature configurations that are the most popular among customers. Similarly, a mobile carrier operator that is about to launch a new set of phone plans may want to discover those plans that would collectively attract the largest number of subscribers.

In this work we develop novel algorithms for two classes of queries involving customer preferences, with practical applications in market research. In the first query type that we consider, we seek to identify customers that may find a product as attractive. We formulate this problem as a *bichromatic reverse skyline query*, and we present a new algorithm, termed as RSA, that outperforms the state-of-the-art algorithm BRS [25] in terms of both I/O and computational cost. Compared to BRS, our RSA algorithm is based on a different processing order, which allows for significant improvements with respect to the *performance*, the *scalability* and the *progressiveness* of returned results when compared to BRS.

Real world applications usually require processing multiple queries efficiently. For example, assume that a mobile carrier operator maintains a database of existing phone plans, customer statistics (i.e., voice usage duration, number of text messages sent, data volume consumed per month) and a list of new phone plans under consideration. We formulate this problem as a new query type, namely the k-Most Attractive Candidates (k-MAC) queries. Given a set of existing product specifications P, a set of customer preferences C and a set of new candidate products Q, the k-MAC query returns the set of k candidate products from Q that jointly maximizes the total number of expected buyers, measured as the cardinality of the union of individual reverse skyline sets (i.e., *influence sets*).

Recent works [12, 15] have independently studied similar problems over 'objective' attributes, i.e. those that have a globally preferred value, such as (zero) price, (infinite) battery life, etc. In such a scenario, the dominance relationships among customer preferences, existing products and candidates can be extracted by executing a single skyline query over the data set. Thereby, these works focus on providing greedy algorithms that determine the most prof-

*Anastasios Arvanitis is currently affiliated with the University of California, Riverside.

CIKM'12, October 29–November 2, 2012, Maui, HI, USA.

itable solution by combining customer sets. In this work we generalize their definition of user preferences, such that we can also handle 'subjective' attributes i.e., those not having a strict order for all users, e.g., as screen size, processor type, operating system etc. For example, for customer A that prefers a portable laptop, a 11" laptop would be more preferable than a 15" one. On the other hand, for a customer B searching for a desktop replacement laptop, the latter model would be more appropriate.

For such attributes, applying methods such as those proposed in [12, 15] requires having extracted the product dominance relationships for all users, since these relations are user-dependent. Thus, we have to execute a dynamic skyline query [14] for each customer, which is prohibitively expensive. Further, applying single point reverse skyline approaches to solve a k-MAC query would require calculating the influence set for each candidate product individually, which is also a very expensive task, especially when handling large data sets. We, thus, propose a batched extension of our RSA algorithm for the k-MAC problem that improves upon processing candidates sequentially by grouping contiguous candidates, exploiting I/O commonalities and enabling shared processing. After extracting the influence set of each candidate product, we also propose an algorithm that greedily calculates the final solution for the k-MAC problem by combining the influence sets of individual candidate products. In brief, the contributions of this paper are:

- We present a novel progressive algorithm, termed as RSA, for (single) reverse skyline query evaluation. Our RSA algorithm scales better in data sets that contain a large number of skyline points (e.g., high-dimensional data), while reporting the first results significantly faster than the state-of-the-art algorithm BRS.
- We develop a batched variant of our RSA algorithm that improves upon processing multiple queries individually, by grouping contiguous candidates, exploiting I/O commonalities and enabling shared processing among similar candidates. We then apply our batched algorithm to solve the k-MAC query. k-MAC generalizes the "k-most demanding products query" of [12] and the "top-k popular products query" of [15] to problems where customer preferences also include subjective attributes.
- We perform an extensive experimental study using both synthetic and real data sets. Our study demonstrates that (*i*) our RSA algorithm outperforms BRS for the reverse skyline query, in terms of I/Os, CPU cost and progressiveness of the output, especially for real data, higher dimensional data, or when the size of the product data set is relatively larger than that of customers, and (*ii*) that our proposed batched algorithm outperforms baseline approaches that process each candidate individually.

2. PRELIMINARIES

2.1 Single Point Reverse Skylines

Consider two sets of points, denoted as P and C, in the same D-dimensional space. We will refer to each point $p \in P$ as a *product*. Each product is a multi-dimensional point, with p_i denoting the product's attribute value A_i. For example, assuming that products are notebooks, the dimensions[1] of p_i may correspond to the notebook's price, weight, screen size, etc. Further, each point $c \in C$ represents a customer's preferred notebook specifications that she would be interested in; we will refer to each point c as a *customer*. Clearly, customers are more interested to the products that are closer to their preferences. In order to capture the preferences of a customer c, we formally define the notion of dynamic dominance.

DEFINITION 1. *(Dynamic Dominance) (from [5]): Let $c \in C$, $p, p' \in P$. A product p dynamically dominates p' with respect to c, denoted as $p \prec_c p'$, iff for each dimension $|p_i - c_i| \leq |p'_i - c_i|$ and there exists at least one dimension such that $|p_i - c_i| < |p'_i - c_i|$.*

Note that this definition can accommodate dimensions with universally optimal values where smaller (larger) values are preferred by simply setting c_i to the minimum (resp. maximum) value of dimension A_i. For example, assuming that lighter notebooks are preferred, we can simply set for all customers $c_{weight} = 0$.

DEFINITION 2. *(Dynamic Skyline) (from [5]): The dynamic skyline with respect to a customer $c \in C$, denoted as $SKY(c)$, contains all products $p \in P$ that are not dynamically dominated with respect to c by any other $p' \in P$.*

Consider a set of existing products $P = \{p_1, p_2, p_3, p_4\}$ and customers $C = \{c_1, c_2, c_3\}$. Figure 1(a) illustrates the dynamic skyline of c_1 that includes notebooks p_2 and p_4 in a sample scenario with 2 dimensions corresponding to the CPU speed and the screen size of a notebook. Points in the shaded areas are dynamically dominated by points belonging to the dynamic skyline of c_1. Since we are interested in the absolute distance between products, a product might dominate other products that belong to different quadrants with respect to a customer. For example, p_1 and p_3 in the upper right quadrant are dynamically dominated by p_2 in the lower right quadrant because p_2 has a CPU speed and a screen size that are both closer to c_1 than the corresponding characteristics of p_1 and of p_3. Figures 1(b) and 1(c) illustrate the dynamic skylines of customers c_2 and c_3 respectively. We now highlight the product's perspective by introducing the definition of *bichromatic reverse skylines*.

DEFINITION 3. *(Bichromatic Reverse Skyline) (from [11]): Let P be a set of products and C be a set of customers. The bichromatic reverse skyline of p, denoted as $RSKY(p)$ contains all customers $c \in C$ such that $p \in SKY(c)$.*

Thus, the bichromatic reverse skyline of a product p contains all customers c that find p as 'attractive'. Henceforth, we refer to the bichromatic reverse skyline of p as the *influence set* of p. Figure 1(d) illustrates the influence sets of products p_1, p_2, p_3 and p_4.

The cardinality of $RSKY(p)$ is a useful metric of the product's impact in the market. We refer to $|RSKY(p)|$ as the *influence score* $\mathrm{IS}(p)$. In our example, $\mathrm{IS}(p_1)=\mathrm{IS}(p_2)=2$ and $\mathrm{IS}(p_3)=\mathrm{IS}(p_4)=1$.

2.2 Influence Region

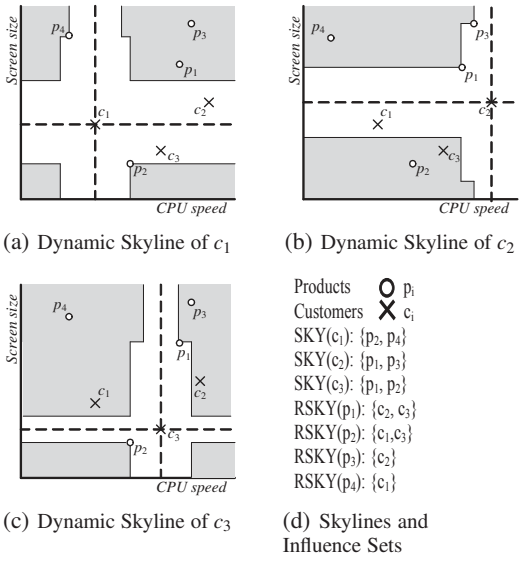
Consider a new product q. The new product partitions the D-dimensional space into 2^D orthants Ω_i, each identified by a number in the range $[0, 2^D - 1]$. Since all orthants are symmetric and we are interested in the absolute distance between products, we can map all products to Ω_0 as illustrated in Figure 2(a). For simplicity, we hereafter concentrate on Ω_0 with respect to a query point q.

For every dynamic skyline point p_i, let $m_i(q)$ be the midpoint of the segment connecting a query point q with p_i. In Figure 2(b) black points m_1, m_2 and m_4 represent the midpoints of p_1, p_2 and p_4 with respect to q. Henceforth, in order to alleviate the complication of maintaining both points and midpoint skylines, whenever we refer to a product p_i we imply the corresponding $m_i(q)$ with respect to q. We also assume that each dynamic skyline point p_i with respect to q is mapped to its midpoint skyline $m_i(q)$ on the fly.

The *influence region* of a query point q, denoted as $\mathrm{IR}(q)$, is the union of all areas not dynamically dominated with respect to q by the midpoint skylines of q. The area in Ω_0 that is not shaded in Figure 2(b) draws the influence region for q. Note that the midpoints themselves belong to the IR, since a tuple cannot dominate itself.

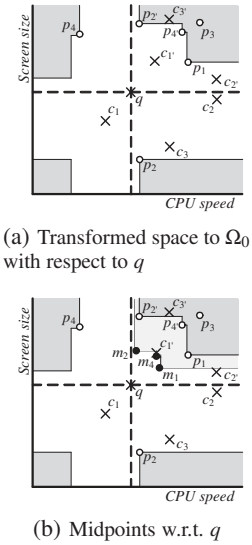
LEMMA 1. *(from [11]) A customer c belongs to the influence set $RSKY(q)$ of q iff c lies inside the influence region of q i.e., $c \in IR(q) \Leftrightarrow c \in RSKY(q)$.*

[1] In the following we will use the terms dimension and attribute interchangeably



(a) Dynamic Skyline of c_1 (b) Dynamic Skyline of c_2 (c) Dynamic Skyline of c_3 (d) Skylines and Influence Sets

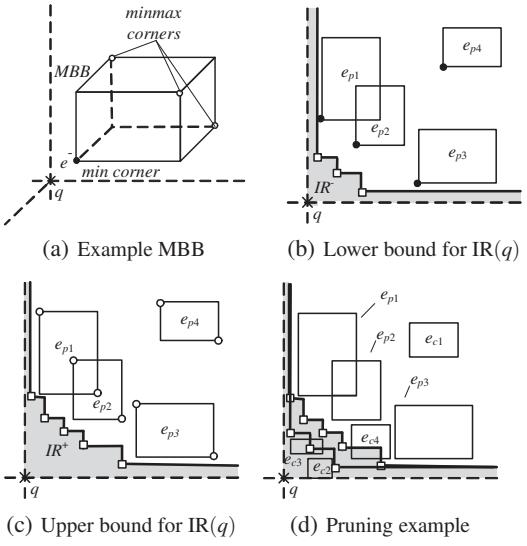
Figure 1: Dynamic Skylines example



(a) Transformed space to Ω_0 with respect to q

(b) Midpoints w.r.t. q

Figure 2: Influence region of q



(a) Example MBB (b) Lower bound for IR(q) (c) Upper bound for IR(q) (d) Pruning example

Figure 3: Influence Regions

Returning to the example of Figure 2(b), notice that only c_2 lies inside IR(q). Therefore, $RSKY(q) = \{c_2\}$.

Hereafter, we assume that all points (either products or customer preferences) are indexed using a multidimensional index (e.g., R-trees, kd-trees etc.); for our presentation we will consider R-trees. Figure 3(a) shows an example minimum bounding box (MBB) e. Inside each MBB e, let *min-corner* $e^-(q)$ denote the point in e having the minimum distance from a query point q. The min-corner dominates the largest possible space. The points that reside in each of the d faces closest to q and are the farthest from the origin q are denoted as *minmax-corners*. Each MBB contains D minmax-corners. Independently of how products within e are distributed, any point in e certainly dominates the area that the minmax-corners do, while at best it dominates the area that the min-corner does.

Given a set of MBBs, we can derive two sets: the set of all min-corners denoted as L and the set of all minmax-corners w.r.t. q denoted as U. Figure 3(b) presents an example, assuming $E_P = \{e_{p_1}, e_{p_2}, e_{p_3}, e_{p_4}\}$, where e_{p_i} denotes a product entry. In Figures 3(b), 3(c) black and hollow circles represent the min-corners and minmax-corners respectively and rectangles represent midpoints.

Continuing the example of Figure 3(b), the grey area represents a lower bound of the actual influence region $\text{IR}^-(q)$ and it is defined as the space not dominated w.r.t. q by any min-corner $l \in L$. Respectively, the grey area in Figure 3(c) represents an upper bound of the actual influence region $\text{IR}^+(q)$, defined as the space not dominated w.r.t. q by any minmax-corner $u \in U$. It follows [25]:

LEMMA 2. *If an entry e_c is dominated by any $u \in U$, i.e. e_c is completely outside $IR^+(q)$, e_c cannot contain any customer inside $IR(q)$. Hence, according to Lemma 1, e_c can be pruned.*

For example, e_{c1} in Figure 3(d) can be pruned because it is completely outside $\text{IR}^+(q)$.

2.3 The BRS Algorithm

In the following we detail the state-of-the-art *Bichromatic Reverse Skyline* (BRS) algorithm [25] that efficiently calculates the influence set of a single query point q. BRS aims at minimizing the I/O cost (*i*) by progressively refining the influence region of q until the influence set of q has been retrieved, (*ii*) by applying Lemma 2 to prune e_c entries that do not contribute to $RSKY(q)$.

BRS uses two indexes, an R-tree T_P on the set of products P and another T_C on the set of customers C. Initially, the algorithm inserts all root entries of T_P (resp. T_C) in a priority queue E_P (resp. E_C) sorted with the minimum Euclidean distance of each entry from q. BRS extracts a set L of all min-corners and a set U of all minmax-corners of $e_p \in E_P$. Further, in order to reduce the number of subsequent dominance checks, BRS calculates the skylines of L and U, denoted as $SKY(L)$ and $SKY(U)$ respectively.

In each iteration BRS expands the entry in E_P with the minimum Euclidean distance from q and updates the current L and U and their skylines $SKY(L)$ and $SKY(U)$. Then, all $e_c \in E_C$ are checked for dominance with $SKY(L)$ and $SKY(U)$. If e_c is not dominated by $SKY(L)$ (i.e. it intersects $\text{IR}^-(q)$), BRS expands e_c as it may contain customers inside IR(q). Returning to Figure 3(d), e_{c3} intersects $\text{IR}^-(q)$; therefore e_{c3} is expanded. In contrast, if a customer entry e_c (such as e_{c1} in Figure 3(d)) is dominated by $SKY(U)$, then e_c can be safely pruned according to Lemma 2. BRS terminates when E_C becomes empty, i.e. the position of all customers either inside or outside IR(q) has been determined.

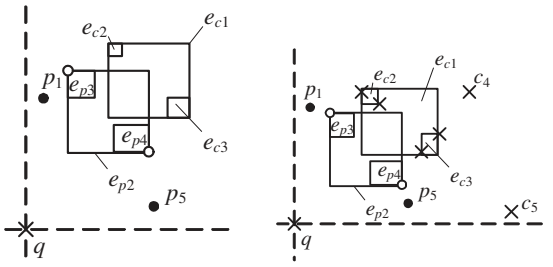
3. EFFICIENT COMPUTATION OF REVERSE SKYLINES

In this section, we detail the drawbacks of BRS and then present a more efficient reverse skyline algorithm, termed RSA.

3.1 BRS Shortcomings

Complexity Analysis. Let p_k, c_k denote the sizes of the currently active entries in E_P and E_C, respectively, after the k-th iteration of the BRS algorithm. The worst-case cardinality of p_k and c_k are $|P|$ and $|C|$ respectively. In each iteration, the BRS algorithm maintains both $SKY(L)$ and $SKY(U)$, two sets with $O(|P|)$ and $O(D|P|)$ entries respectively, where D is the dimensionality of the data set. BRS then checks for dominance each entry in E_P and E_C with both $SKY(L)$ and $SKY(U)$. Thus, each iteration entails $O(D|P| \times (|P| + |C|))$ dominance checks, which require a total of $O(D^2|P| \times (|P| + |C|))$ comparisons, since each dominance check requires $O(D)$ comparisons.

Clearly, the processing cost of BRS depends on the size of the intermediate upper and lower skyline sets. [2] shows that for uniformly distributed data the size of the skyline set is $\Theta(\frac{(ln|P|)^{D-1}}{D!})$. Thus, for larger data sets or higher dimensional data the processing cost of maintaining $SKY(L)$ and $SKY(U)$ becomes prohibitively expensive. Our experimental evaluation (Section 6) confirms that



(a) BRS will expand e_{p_2} entailing an unnecessary I/O

(b) RSA will avoid accessing e_{p_2} by pruning c_4 with p_1

Figure 4: Processing order and I/O accesses

BRS is impractical for $|P| \geq 10^6$ or $D \geq 4$. Motivated by the above analysis, we introduce a more efficient and scalable reverse skyline algorithm, which eliminates the dependency on the $SKY(L)$ and $SKY(U)$ sets, thus being able to handle high dimensional data, or, in general, data where the size of skyline points is large.

Processing Order. BRS performs a synchronous traversal on the T_P and T_C indexes, which are built on product and customer points, respectively, following a monotonic order based on the Euclidean distance of e_p entries from q. This processing order ensures that the number of I/Os on T_P is minimized. However, in terms of the total I/Os, BRS might perform some unnecessary I/Os. Figure 4(a) illustrates one such scenario, where the nodes e_{p_2} and e_{c_1} have not been yet expanded.[2] BRS would proceed by expanding e_{p_2}, revealing e_{p_3} and e_{p_4}. Unfortunately, e_{c_1} is not affected by this refinement and it still has to be accessed. On the other hand, if we first expand e_{c_1}, this operation would reveal e_{c_2} and e_{c_3}, which can be pruned by p_1 and p_5 respectively, eliminating the need to access e_{p_2}. Clearly, in this scenario the I/O access on e_{p_2} was redundant. In order to avoid such redundant I/Os, our RSA follows a visiting strategy that is primarily based on the *tree level* of customer entries, which, as confirmed in our experiments, results in fewer total I/Os.

Progressiveness. BRS iteratively refines $\text{IR}^-(q)$ and reports the customer points that lie inside $\text{IR}(q)$. In order to retrieve the first reverse skyline point, several iterations of BRS may be required, which is undesirable for applications that require a quick response containing only a fraction of the output, or if the complete output is not useful (e.g., if it contains too many results). We, thus, seek to develop an algorithm that reports the first results faster than BRS.

3.2 The RSA Algorithm

We now present our *Reverse Skyline Algorithm* (RSA), which aims to address the shortcomings outlined above.

Data Structures Used and Basic Intuition. The RSA algorithm:

- Does not require the maintenance of the $SKY(L)$ and $SKY(U)$ sets and is, thus, less expensive in terms of processing cost.
- Checks one customer entry per iteration following a visiting strategy based on the entry's tree level (primary sort criterion) and Euclidean distance from q (secondary sort criterion).
- Accesses a product entry only if it is absolutely necessary in order to determine if a customer point belongs to $RSKY(q)$.

RSA maintains the following data structures for its operation:

- A priority queue E_P on the set of product entries
- A priority queue E_C on the set of customer entries
- A set $SKY(q)$ with the currently found midpoint skylines

The two priority queues are sorted based on a dual sorting criterion: primarily, based on the tree level of the stored entries and, subsequently, using the Euclidean distance of each entry from q.

[2] Note that with e_p we actually represent the respective midpoints w.r.t q.

Algorithm 1: RSA

Input: q a query point, T_P R-tree on products, T_C R-tree on customers, $E_P(q)$ priority queue on products, $E_C(q)$ priority queue on customers
Output: $RSKY(q)$ reverse skylines of q
Variables: $SKY(q)$ currently found midpoint skylines of products w.r.t. q

```
1  begin
2      SKY(q) := ∅; RSKY(q) := ∅;
3      while E_C ≠ ∅ do
4          dominated := false;
5          E_C(q).pop() → e_c;
6          if dominated(e_c, SKY(q)) then
7              dominated := true; continue;
8          if e_c is a non-leaf entry then
9              Expand e_c, insert children entries in E_C(q);
10         else
11             foreach e_p ∈ E_P(q) do
12                 midpoint(e_p, q) → m;
13                 if e_c is dominated by m then
14                     if e_p is a leaf entry then
15                         if (dominated(m, SKY(q)) == false) then
16                             SKY(q).push(m);
17                         dominated := true; break;
18                     else
19                         Expand e_p, insert children entries in E_P(q);
20                     E_P(q).remove(e_p);
21             if (dominated == false) then
22                 RSKY(q).push(e_c);
23     return RSKY(q);
```

Thus, leaf entries are given higher priority and are processed first, while the examination of non-leaf entries is postponed as much as possible. By first processing all leaf e_c entries, the algorithm may reveal a midpoint skyline, which will be subsequently used to prune a non-leaf e_c based on Lemma 2, thus avoiding an access on T_C. The same intuition holds for e_p entries as well; an already found midpoint skyline can be used to prune dominated non-leaf product entries, since these entries will not contribute to the skyline. Further, whenever an e_c entry is checked for dominance with E_P, first all leaf e_p entries will be examined. As long as no leaf e_p dominates e_c, only then will RSA proceed to expand the nearest to q non-leaf e_p entry. This change in the visiting order of E_P reduces the number of accesses on T_P as well. For instance, in Figure 4(b) BRS would access e_{p_2} that has the minimum Euclidean distance from q. In contrast, RSA will use p_1 to determine that c_4 does not belong to $RSKY(q)$, hence avoiding the access of e_{p_2}.

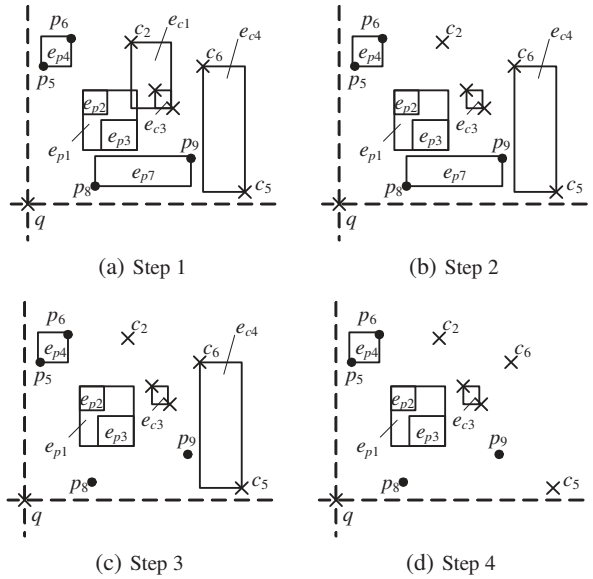
Algorithm Description. The RSA algorithm is presented in Algorithm 1. Initially, RSA inserts all root entries of T_P (resp. T_C) in the priority queue E_P (resp. E_C). Further, RSA maintains a set $SKY(q)$ of the currently found midpoint skylines which are used for pruning based on Lemma 1. RSA proceeds in iterations. In each iteration RSA extracts the entry in E_C having the minimum key from q (Line 5) and checks the following pruning conditions:

1. If e_c is dominated by any point that belongs to the currently found midpoint skylines $SKY(q)$, e_c can be removed from E_C based on Lemma 1 (Lines 6-8).

2. Otherwise, if e_c is a non-leaf entry (Line 9), e_c is expanded and child nodes are inserted into E_C (Line 10).

3. Else, for all e_p entries in E_P (Lines 12-22):

- If e_c is dominated by the midpoint of a leaf entry $e_p \in E_P$ (Line 15), then e_c can be removed from E_C, based on Lemma 1, and the midpoint of e_p is inserted into $SKY(q)$ (Line 17)).
- Else if e_c is dominated by the midpoint of the *min-corner* e_p^- of a non-leaf $e_p \in E_P$ (Line 20), e_p is expanded and its children entries are inserted into E_P (Line 21).



(a) Step 1 (b) Step 2

(c) Step 3 (d) Step 4

Figure 5: Running example of the RSA algorithm

Finally, if e_c has not been pruned by any of the above conditions (Line 23), then e_c is a reverse skyline point and can be at that stage reported as a result (Line 24). The RSA algorithm terminates when E_C becomes empty and then $RSKY(q)$ is returned (Line 25).

Example. We illustrate the execution of RSA using the running example depicted in Figure 5. At the beginning, $E_P(q) = \{e_{p_7}, e_{p_1}, e_{p_4}\}$ and $E_C(q) = \{e_{c_1}, e_{c_4}\}$ (sorted by their distance from q). In the first iteration, RSA will examine e_{c_1} that has the minimum distance from q. Since it is a non-leaf entry, RSA will expand e_{c_1} (Line 10) and it will insert child nodes c_2 and e_{c_3} into $E_C(q)$ (see Figure 5(b)). Now $E_C(q) = \{c_2, e_{c_4}, e_{c_3}\}$ and RSA selects to examine c_2. Since the current skyline is empty, c_2 is not dominated by any product entry; hence RSA will proceed by checking if c_2 is dominated by any product entry contained in $E_P(q)$. c_2 is dominated by the min-corner of the first entry in $E_P(q)$, i.e. e_{p_7} (Line 14). In order to determine if there actually exists a point inside e_{p_7} that dominates c_2 w.r.t. q, RSA will expand e_{p_7} (Line 21), pushing its child nodes p_8 and p_9 inside $E_P(q)$, thus $E_P(q) = \{p_8, p_9, e_{p_1}, e_{p_4}\}$ (see Figure 5(c)). Now, RSA discovers that c_2 is dominated by p_8, which is marked as a skyline point (Line 17) and c_2 is discarded. In the next iteration, RSA selects to examine e_{c_4}, which has the minimum distance from q. e_{c_4} is not dominated by any currently found skyline point and since it is a non-leaf entry, it is expanded and child nodes c_5 and c_6 are inserted into $E_C(q)$ (Line 10) (see Figure 5(d)). Now we have $E_C(q) = \{c_5, c_6, e_{c_3}\}$. Next, RSA will examine c_5. Since c_5 is not dominated by any product entry, c_5 is reported as a reverse skyline result (Line 24). Now RSA examines c_6 which is already dominated by a currently found skyline point, i.e. p_8, (Line 6), hence it is discarded. Finally, e_{c_3} is examined. Similarly, e_{c_3} is dominated by p_8 and it is also pruned. Since $E_C(q)$ is now empty, RSA terminates and outputs c_5 as the final answer.

Complexity and Progressiveness Analysis. RSA requires at most $|C|$ iterations (one for each customer), although in fact several e_c entries will be pruned by $SKY(q)$ (Line 6). Each iteration entails a dominance check with (*i*) the currently found midpoint skyline $SKY(q)$, and (*ii*) all product entries currently in E_P, both having $O(|P|)$ worst-case cardinality. Overall RSA requires $O(|P||C|)$ dominance checks, or $O(D|P||C|)$ comparisons.

With respect to progressiveness, recall that RSA will first examine leaf customer entries that have the minimum Euclidean distance from the query point q (based on the dual sorting scheme on E_C). In other words, the very first iterations of RSA concern customer entries that are very close to q. Intuitively, the closer to q a customer is, the more likely that q will not be dominated by any product w.r.t. the examined customer. Hence, customers that will be examined in the first iterations tend to have a higher probability of belonging to $RSKY(q)$. Further, since the first entries to be examined are actual points (not MBBs), the first iterations will not involve e_c expansions (Line 10), which are expensive in terms of processing cost. Thus, the first iterations will be faster than subsequent ones. Overall, RSA typically reports the first results in just a few iterations. In contrast, recall that BRS requires several iterations in order to adequately refine the influence regions, such that the first reverse skylines have been determined. Our experimental study (Section 6) verifies the superiority of RSA in terms of progressiveness compared to the BRS algorithm.

4. K-MAC QUERIES

We now present the *k*-Most Attractive Candidates (k-MAC) query, which serves as a motivating example that demonstrates the need to develop a batch processing algorithm for computing several reverse skyline queries. The k-MAC query is a slight generalization of the problems studied in [15, 12] for the case when the customer preferences also include subjective dimensions. We first present a motivating scenario, which highlights the usefulness of k-MAC queries. We then present the definition of the k-MAC query.

Motivating Scenario. A laptop manufacturer wants to produce k new notebooks, among a set of feasible alternative configurations Q proposed by the engineering department. The manufacturer needs to consider three sets: (*i*) the existing competitor products P, (*ii*) the set of customers' preferred specifications C, and (*iii*) a set of candidate products Q. We will refer to each $q \in Q$ as a *candidate*. The goal of the manufacturer is to identify the specifications that are expected to *jointly* attract the largest number of potential buyers. Note that this is different than simply selecting the k products that are the most attractive individually, since it does not make much sense to select products that seem attractive to the same set of customers.

Problem Definition. We first define the *joint influence set* for a set of candidates Q. We then define the notion of the *joint influence score* and introduce the *k-Most Attractive Candidates (k-MAC)* query.

DEFINITION 4. *(Joint Influence Set): Given a set of products P, a set of customers C and a set of candidates Q, the joint influence set of Q, denoted as $RSKY(Q)$, is defined as the union of individual influence sets of any $q_i \in Q$: $RSKY(Q) = \bigcup_{q_i \in Q} RSKY(q_i)$*

Following the above definition, the *joint influence score* $\mathrm{IS}(Q)$ for a set of candidates Q is equal to the size of the joint influence set of Q, $|RSKY(Q)|$. We now introduce the *k-Most Attractive Candidates* (k-MAC) query as follows:

DEFINITION 5. *(k-Most Attractive Candidates (k-MAC) query): Given a set of products P, a set of customers C, a set of candidates Q and an integer $k > 1$, determine the subset $Q' \subseteq Q$, such that $|Q'| = k$ and the joint influence score of Q', $IS(Q')$, is maximized.*

Note that several candidates might be interesting for the same customer. Additionally, we emphasize that for evaluating a k-MAC query each candidate $q \in Q$ is considered separately from other candidates and only with respect to existing products. In other words, intra-candidate dominance relations are not taken into account for k-MAC queries. This is consistent with a real-world setting where a manufacturer is interested to compare their product portfolio only with respect to the competition. We discuss how we resolve ties at the end of this section where we present a greedy algorithm that computes an approximate solution for the k-MAC problem.

Unlike recent works [15, 12] that targeted similar problems assuming only 'objective' attributes, i.e. those having a globally preferred value (such as zero price, infinite battery life, etc.), k-MAC can handle cases where customer preferences are expressed over 'subjective' dimensions (e.g., screen size, processor type). This generalisation is possible because the attractiveness of each candidate products is computed based on the size of their bichromatic influence set. Moreover, while our focus is on efficiently computing the influence sets of multiple candidate products, the emphasis of [15, 12] is on the selection of the proper candidates *after* the dominance relationships among products have been determined.

A Greedy Algorithm. Unfortunately, processing k-MAC queries is non-trivial. This problem can be reduced to the more general *maximum k-coverage* problem. Thus, even if we consider the much simpler problem where all the influence sets of all candidates have been computed, an exhaustive search over all possible k-cardinality subsets of Q is NP-hard. Based on the complexity of computing the subset of k products, we now seek an efficient, greedy algorithm for this problem. Our solution is based on the generic k-stage covering algorithm provided in [8], developed for finding efficient approximate solutions to the maximum k-coverage problem.

LEMMA 3. *(from [8]) k-stage covering algorithm returns an approximate solution to the maximum k-coverage problem that is guaranteed to be within a factor* $1-1/e$ *from the optimal solution.*

We now show how we can adapt the k-stage covering algorithm for the k-MAC problem. kGSA (k-stage Greedy Selection Algorithm) takes as input a set of candidate products Q and their associated influence sets and returns a set $Q' \subseteq Q$, $|Q'| = k$ that contains the candidates which formulate a $(1-1/e)$-approximate solution to the k-MAC query. kGSA proceeds in iterations, by adding one candidate into Q' during each iteration. All candidates are examined at each iteration, and kGSA selects the one that, if added in Q', results in the largest increase of the joint influence score of Q'. In case multiple candidates contribute equally to the increase of $\mathrm{IS}(Q')$, kGSA applies a second criterion; it selects the candidate with the minimum sum of distances from its respective reverse skylines (customers). The intuition is that a candidate product that is closer to a user's preferences would more likely increase user satisfaction. kGSA terminates after k iterations and returns Q'.

5. REVERSE SKYLINE PROCESSING FOR MULTIPLE QUERY POINTS

Solving the k-MAC problem requires processing all candidates, in order to first determine their influence sets. The kGSA algorithm can then be used to solve the k-MAC problem.

A straightforward way to process multiple candidates would be to apply either BRS or RSA for each candidate individually. However this approach is very inefficient in terms of I/O accesses, because it requires accessing each entry e_p (e_c) several times, if e_p (e_c) appears in the priority queues of more than one candidate.

Our bRSA algorithm. We now introduce our bRSA algorithm, which aims at eliminating the drawbacks of the baseline approach by exploiting I/O commonalities and by offering shared processing among candidates. bRSA utilizes in its core the RSA algorithm that we presented in Section 3. Note that, apart from k-MAC queries, bRSA can be applied for any query type that requires joint processing of multiple reverse skyline queries.

bRSA efficiently processes candidates in parallel, by grouping them in batches, in such a way that grouped candidates benefit by the processing of other group members. A primary goal of bRSA is to save duplicate I/O accesses, by using entries that have been expanded during an iteration of the RSA subcomponent for one group member, in order to prune entries that appear in the local priority queues of other group members as well. In particular, whenever an entry e_x is expanded, all local priority queues in which e_x appears are appropriately updated. Hence, each disk page is accessed only once per batch. Additionally, in order to further optimize the processing of group members, bRSA maintains a list of currently found product points, that will expectedly have large pruning potential for other group members, based on Lemma 1. We will refer to these points as *vantage points* and we will explain their use in the following where we discuss bRSA execution in detail.

Note that we cannot safely assume that all the necessary data structures that bRSA utilizes (local priority queues, skyline sets for each candidate, list of vantage points etc.) will actually fit in main memory. Based on the memory capabilities of the hardware and worst case estimates of the amount of customer and product entries in E_P and E_C, let us assume that G candidates (where $G \ll |Q|$) fit in main memory and can be simultaneously processed. Using worst case estimates does not have a severe impact in the performance of bRSA; in fact, as we demonstrate experimentally, it is better to keep G to fairly modest values (i.e., up to 10 candidates). Larger batch sizes may result in increasing processing cost for the maintenance of local priority queues and significantly more dominance checks which gradually eliminates the benefit from shared processing.

Candidates in proximity in the multidimensional space are more likely to benefit from shared processing. Hence, as a preprocessing step, bRSA partitions the candidate set into $\lceil |Q|/G \rceil$ batches based on a locality preserving hashing method, such as the Hilbert space filling curve.[3] Then, bRSA picks one candidate at a time in a round robin fashion (Line 5), and executes a single iteration of a modified version of the RSA algorithm for that candidate, termed Batch-RSA. Batch-RSA extends RSA to be efficiently used on a batch setting. We now present the differences of Batch-RSA compared to its single point counterpart. First, whenever an entry e_x is expanded, all local priority queues in which e_x appears are appropriately updated. Further, when a leaf product entry, say p_i, is discovered (Line 12), the algorithm decides whether p_i should be inserted to a buffer H_P that contains vantage points, i.e. those that will be used for pruning by other candidates (Line 17). Intuitively, product points that reside closer to a candidate, will dominate the largest possible space and their pruning power will be maximized. Thus, we implemented H_P as a priority queue on the minimum Euclidean distance, among the candidates inside the batch. If H_P is full, the most distant point in H_P, is replaced with p_i. Vantage points (essentially their respective midpoints) are used additionally to skyline points when checking each customer entry for dominance (second condition in OR clause of Line 5), hence avoiding some of the subsequent I/Os.

6. EXPERIMENTAL EVALUATION

All algorithms examined in our experiments were implemented in C++ and executed on a 2.0 GHz Intel Xeon CPU with 4 GB RAM running Debian Linux. The code for the BRS algorithm was thankfully provided to us by the authors of [25].

6.1 Experimental Setup

We used a publicly available generator [1] in order to construct different classes of synthetic data sets, based on the distribution of the attributes' values; i.e., uniform (UN), anti-correlated (AC) and correlated (CO). Due to space limitations, in the following we plot the results primarily for uniform (UN) data sets. Experiments involving AC and CO data, as well as combinations among them (e.g., uniformly distributed products and anti-correlated customers)

[3] Other clustering techniques might also be applicable

Algorithm 2: bRSA

Input: Q a set of candidates, T_P R-tree on products, T_C R-tree on customers
Variables: $E_P(q_i)$ priority queue on products for q_i, $E_C(q_i)$ priority queue on customers for q_i, $RSKY(q_i)$ reverse skylines for q_i, $SKY(q_i)$ midpoint skylines of q_i, G_j batches with $|G_j| = G$

```
1  begin
2      partition Q into ⌈|Q|/G⌉ batches → G_j;
3      foreach G_j do
4          while (RSKY(q_i) for all q_i ∈ G_j have not been found) do
5              selectCandidate → q_i;
               /* Process q_i until IS(q_i) has been
                  completely determined                    */
6              if E_C(q_i) ≠ ∅ then
7                  Batch-RSA (q_i, G_j, T_P, T_C, E_P(q_i), E_C(q_i), RSKY(q_i),
                   SKY(q_i), H_P);
```

Function Batch-RSA

Input: G a group of candidates, T_P R-tree on products, T_C R-tree on customers, $E_P(q_i)$ priority queue on products for q_i, $E_C(q_i)$ priority queue on customers for q_i, $RSKY(q_i)$ reverse skylines of q_i, $SKY(q_i)$ midpoint skylines of q_i, H_P priority queue on product leaf entries (vantage points)
Output: $RSKY(q_i)$ reverse skylines of q_i

```
1  begin
2      while E_C(q_i) ≠ ∅ do
3          dominated := false;
4          E_C(q_i).pop() → e_c;
5          if dominated(e_c, SKY(q_i)) OR dominated(e_c, H_P) then
6              dominated := true; continue;
7          if e_c is a non-leaf entry then
8              Expand e_c for all relevant q_i, insert children into E_C(q_i);
9          else
10             foreach e_p ∈ E_P(q_i) do
11                 midpoint(e_p, q_i) → m;
12                 if e_c is dominated by m then
13                     if e_p is a leaf entry then
14                         if (dominated(m, SKY(q_i)) == false) then
15                             SKY(q_i).push(m);
16                         H_P.push(e_p);
17                         dominated := true; break;
18                     else
19                         Expand e_p for all relevant q_i, insert children into
                           E_P(q_i);
20                     E_P(q_i).remove(e_p);
21             if (dominated == false) then
22                 RSKY(q_i).push(e_c);
23     return RSKY(q_i);
```

generally follow similar trends. We also evaluated our algorithms on two real world data sets. The NBA data set (NBA) consists of 17,265 5-dimensional points, representing the average values of a player's annual performance with respect to the number of points scored, rebounds, assists, steals and blocks. The household data set (HOUSE), consists of 127,930 6-dimensional points, representing the percentage of an American family's annual income spent on 6 types of expenditure: gas, electricity, water, heating, insurance, and property tax. In order to generate customer and candidate sets, we added Gaussian noise to actual points. For both synthetic and real data sets we normalized all attribute values to [0,10000] and for each data set, we built an R-tree with a page size equal to 4KB.

We compared the performance of our RSA and bRSA algorithms with the state-of-the-art BRS algorithm for evaluating both reverse skyline and k-MAC queries. For BRS and RSA, we measured the total CPU time and I/O operations required for processing *(i)* a workload of $|Q|$ reverse skyline queries, and *(ii)* a k-MAC query given an input of $|Q|$ candidate products. The bRSA algorithm applies only for the k-MAC query. In particular, we measured:

- The number of I/Os (separately on product and customer entries). For each data set, one memory buffer equivalent to 100 pages (12.5% of the data set size) was allocated for caching, following a Least Recently Used (LRU) cache replacement policy.
- The time spent on CPU.
- The total query processing time, consisting of the time spent on CPU plus the I/O cost, where each random page access was penalized with 1 millisecond.

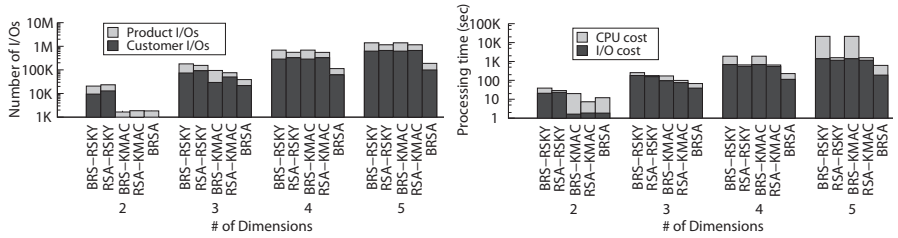
Recall that for the reverse skyline query type, both BRS and RSA process query points sequentially. For evaluating k-MAC queries, we modified both algorithms by adding *(i)* a preprocessing step that presorts candidates based on their Hilbert hash value, and *(ii)* a final step that greedily outputs the best candidates using our kGSA algorithm. In our experiments the measured processing time required for both steps was negligible compared to the time required for the algorithm execution. Further, it is important to emphasize that none of the algorithms is affected by the value of k, since they first have to determine the influence sets of all candidates, and then greedily select the optimal k-subset based on kGSA.

In each experiment we vary a single parameter while setting the remaining to their default values. The default values for both product and customer data set cardinalities were set to 100,000, the default data dimensionality was set to 3, the default domain range of each attribute was [0, 10000], the default batch size was set to 10, and the default buffer size was set to 12.5% of the data set size.

6.2 Experimental Results

Sensitivity Analysis vs. Data Dimensionality. We first vary the dimensionality of the data sets from 2 to 5 and examine the performance of all algorithms. Figures 6(a)-6(b) show the results for the number of I/Os and the total processing time, respectively, in logarithmic scale. The corresponding numbers for the BRS and RSA algorithms are also presented, for clarity, in Figure 6(c). As expected (refer to the complexity analysis in Section 3), BRS becomes prohibitively expensive for data with more than 3 dimensions. In particular, BRS requires 3.35 times more CPU time than RSA even in 2 dimensions, and is about 46 times slower in terms of CPU time, and 13.5 times slower in terms of total processing time in the 5-dimensional data set. Figure 6(k) shows an analogous behaviour of the algorithms in anti-correlated data. We also experimented with higher dimensionalities, e.g., for $D = 6$, BRS took $\sim$15 hours to finish, whereas RSA terminated in 20.2 minutes. However, we did not include these results in the plots due to space limitations. Our experiments with real data sets show that BRS is impractical for higher dimensions, which justifies our motivation for a more efficient reverse skyline algorithm. It is important to notice that in higher dimensionalities the I/O cost of BRS is dominated by the CPU cost (note that Figures 6(a)-6(b) are in logarithmic scale). To understand why BRS escalates poorly with D recall that the sizes of $SKY(L)$ and $SKY(U)$, which are maintained by BRS, increase rapidly with the data dimensionality [2]. Finally, our bRSA algorithm achieves significant performance gains with respect to both BRS and RSA in all settings. Note that our remaining sensitivity analysis using synthetic data sets, utilizes a modest value ($D = 3$), which is a favorable setting for BRS. Obviously, the benefits of our algorithms over BRS were significantly more in higher dimensions.

Sensitivity Analysis vs. Data Set Size. We then perform a sensitivity analysis with respect to the size of the product data set. Notice the different behavior of the two algorithms with respect to the type of I/Os (Figure 6(d)), due to the different visiting orders followed; generally BRS entails more accesses on the products index, whereas RSA requires more customer I/Os. In terms of I/Os, RSA exhibits similar performance with BRS in the case where the product and customer data have the same size (100K both). However, as the number of products increases, the strategy followed



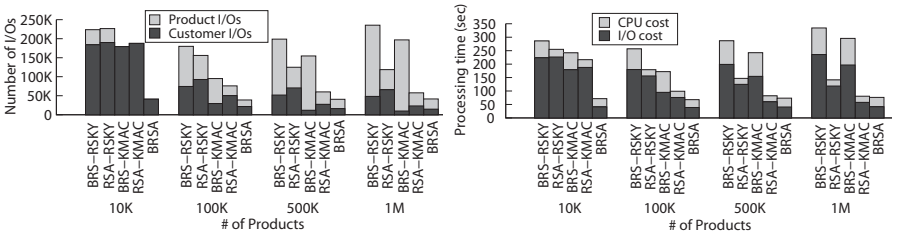
(a) I/Os vs. Dimensionality (UN)

(b) CPU cost vs. Dimensionality (UN)

	2-D data		3-D data		4-D data		5-D data	
	BRS	RSA	BRS	RSA	BRS	RSA	BRS	RSA
I/O cost (sec)	20.7	23.3	179.9	156.0	687.6	555.0	1404	1161
CPU cost (sec)	18.4	5.5	76.9	23.2	1222.6	106.2	20042	428
Total cost (sec)	39.1	28.8	256.8	179.2	1910.2	661.2	21446	1589

	2-D data	3-D data	4-D data	5-D data
	BRS / RSA	BRS / RSA	BRS / RSA	BRS / RSA
I/O cost ratio	0.89	1.15	1.24	1.21
CPU cost ratio	3.35	3.32	11.51	46.81
Total cost ratio	1.36	1.43	2.89	13.50

(c) RSA & BRS costs vs. Dimensionality (UN)



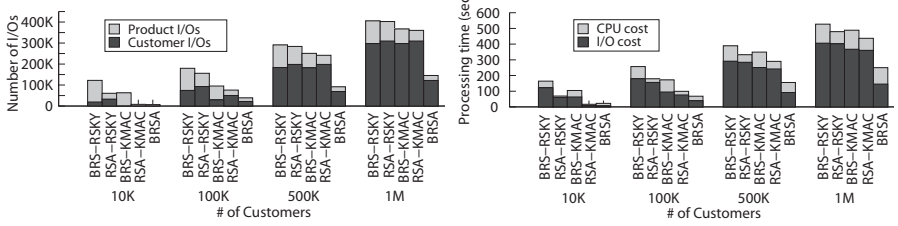
(d) I/Os vs. $|P|$ (UN)

(e) CPU cost vs. $|P|$ (UN)

	10K		100K		500K		1M	
	BRS	RSA	BRS	RSA	BRS	RSA	BRS	RSA
I/O cost (sec)	223.6	226.5	179.9	156.0	198.9	125.0	235.5	118.5
CPU cost (sec)	62.7	28.6	76.9	23.2	87.9	22.0	98.8	22.9
Total cost (sec)	286.3	255.1	256.8	179.2	286.8	147.0	334.3	141.4

	10K	100K	500K	1M
	BRS / RSA	BRS / RSA	BRS / RSA	BRS / RSA
I/O cost ratio	0.99	1.15	1.59	1.99
CPU cost ratio	2.19	3.32	3.99	4.32
Total cost ratio	1.12	1.43	1.95	2.36

(f) RSA & BRS costs vs. $|P|$ (UN)



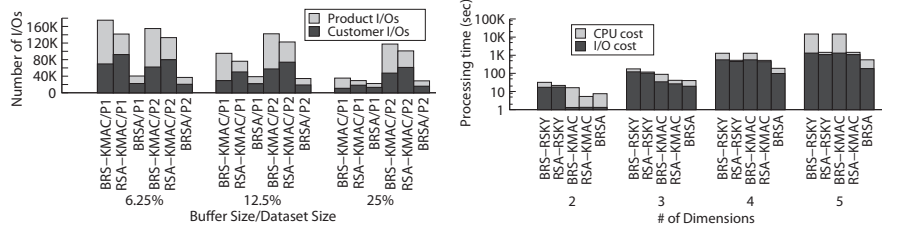
(g) I/Os vs. $|C|$ (UN)

(h) CPU cost vs. $|C|$ (UN)

	10K		100K		500K		1M	
	BRS	RSA	BRS	RSA	BRS	RSA	BRS	RSA
I/O cost (sec)	122.4	60.6	179.9	156.0	291.0	284.1	405.8	402.4
CPU cost (sec)	41.6	8.5	76.9	23.2	98.1	48.1	121.5	77.0
Total cost (sec)	164.0	69.1	256.8	179.2	389.1	332.2	527.3	479.4

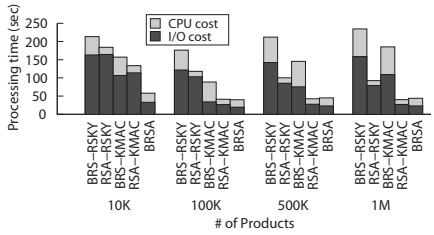
	10K	100K	500K	1M
	BRS / RSA	BRS / RSA	BRS / RSA	BRS / RSA
I/O cost ratio	2.02	1.15	1.02	1.01
CPU cost ratio	4.91	3.32	2.04	1.58
Total cost ratio	2.37	1.43	1.17	1.10

(i) RSA & BRS costs vs. $|C|$ (UN)



(j) I/Os vs. Cache Size (UN)

(k) CPU cost vs. D (AC)



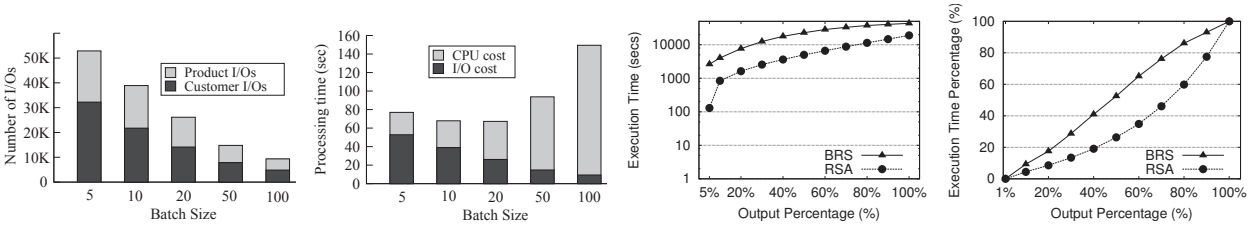
(l) CPU cost vs. $|P|$ (AC)

Figure 6: Experiments with Synthetic Data

by RSA proves to be more efficient in terms of the total I/O accesses required. Moreover, w.r.t. the processing cost (Figure 6(e)), RSA is significantly faster than BRS, and scales better as $|P|$ grows larger. Again, the corresponding numbers for the BRS and RSA algorithms are also presented, for clarity, in Figure 6(f). Finally, our bRSA algorithm is the most efficient algorithm for the case of k-MAC queries remaining essentially unaffected by the size of the product data set. Figure 6(l) shows an analogous behaviour of the algorithms in anti-correlated data, with the times being slightly smaller, as shown by the scale of the y-axis. In Figures 6(g)-6(h) we then plot the I/O and CPU costs when varying the size of the customer data set. As illustrated, RSA and BRS require roughly the same number of I/Os for large numbers of customers. This is predictable since RSA processes one customer entry per iteration, i.e., the number of iterations required by RSA is $O(|C|)$. Therefore, in the case when $|C|$ is much larger than $|P|$, the visiting strategy followed by BRS would be a more reasonable choice. However, even in this worst case scenario, RSA exhibits better overall performance than BRS algorithm, due to the significant lower processing cost (Figures 6(h)-6(i)). Again, our bRSA algorithm is notably faster than both single point algorithms.

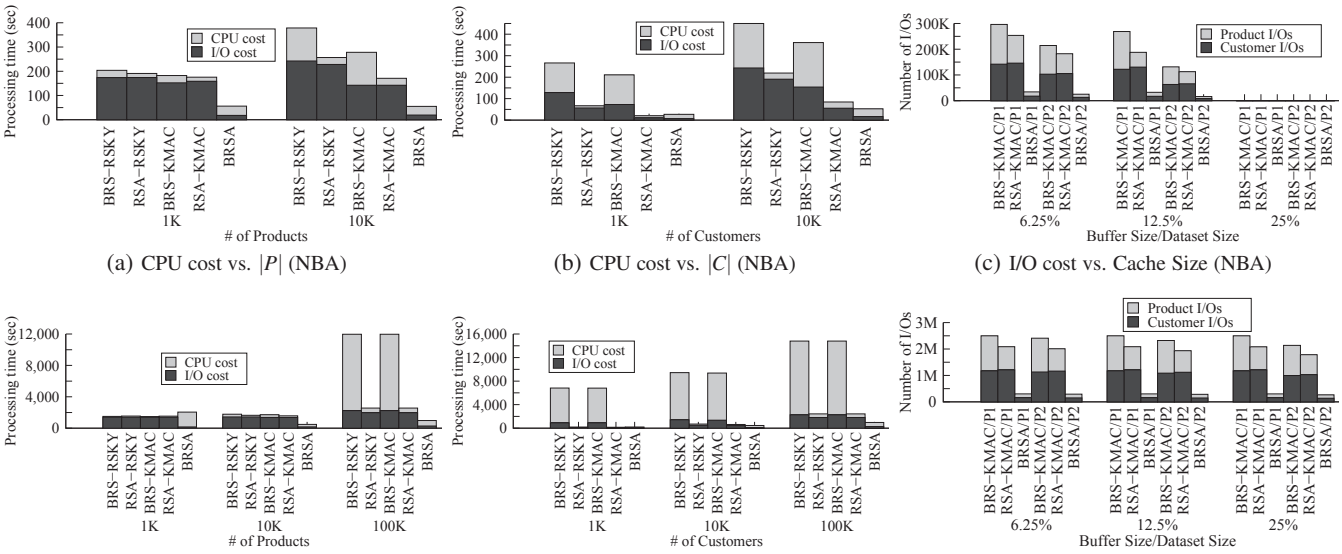
Sensitivity Analysis vs. Memory Size. For this experiment we compare the number of page accesses required by each algorithm with respect to the memory size allocated for caching. We varied cache (buffer) size from 50 pages (corresponding to 6.25% of the data set size) up to 200 pages (25% of available memory). We also experimented with two different cache replacement policies. For the first policy, namely $P1$, we followed a LRU strategy. Additionally, motivated by the intuition that entries with higher levels in the R-tree will be accessed more frequently, we also used a buffer that maintains pages in descending order of their tree level ($P2$). Figure 6(j) plots the number of I/Os required for different cache sizes and cache replacement policies. As depicted, regardless of the memory size and strategy used, both RSA and bRSA algorithms are more efficient in terms of disk accesses. Further, notice that LRU was slightly more efficient for the cache size that we used in our default scenario (12.5% of the data set size).

Sensitivity Analysis vs. Batch Size. We then investigate the performance of our bRSA algorithm with respect to the batch size G. We set each batch to contain from 5 to 100 candidates and plotted the results in Figures 7(a)-7(b). As expected, larger batch sizes result to fewer total I/O operations, since more pruning can be



(a) I/Os vs. Batch Size (UN) (b) CPU cost vs. Batch Size (UN) (c) CPU cost vs. number of reported results (UN) (d) CPU cost/Total cost (%) vs. number of reported results (UN)

Figure 7: Varying the Batch Size (left), Progressiveness of Reported Results (right)



(a) CPU cost vs. $|P|$ (NBA) (b) CPU cost vs. $|C|$ (NBA) (c) I/O cost vs. Cache Size (NBA)

(d) CPU cost vs. $|P|$ (HOUSE) (e) CPU cost vs. $|C|$ (HOUSE) (f) I/O cost vs. Cache Size (HOUSE)

Figure 8: Experiments with Real Data

shared among candidates, whereas the processing cost increases. Interestingly, when the batch size becomes larger than a threshold, the total processing cost (cost of disk accesses plus CPU time) increases, due to the growing cost of maintaining all local priority queues and the significantly more dominance checks required thereof. As showcased by the experiments, keeping fairly small batch sizes ($\sim$10 candidates) maximizes the efficiency of bRSA.

Progressiveness. Finally, we compare the progressiveness of RSA and BRS algorithms on a workload consisting of $|Q|$ reverse skyline queries. x-axis represents the percentage of reverse skyline results found so far compared to the total influence score. y-axis plots the time required to report the corresponding percentage of results, both in absolute time (Figure 7(c)) and as a percentage of the total time spent (Figure 7(d)). Both figures demonstrate that RSA is notably more progressive than BRS, especially for reporting the first query results. In particular, RSA outputs the first 5% of the reverse skylines in 1/10th of the time needed by BRS, which can be particularly important for applications that require a quick response or when the complete output is not useful.

Experiments with Real Data. Figures 8(a)-8(c) and 8(d)-8(f) report our experimental findings on the NBA and HOUSE data sets respectively. The results are in accordance with our experiments on synthetic data sets. Moreover, the performance gains achieved by RSA are higher in real data sets (especially vs. $|C|$), partly due to the higher data dimensionality (5 and 6 dimensions respectively), which results to more points belonging to the influence set (on average 196 points in HOUSE data set vs 11 points in UN data set).

7. RELATED WORK

Market research is a systematic, objective collection and analysis of data about a particular target market by taking into account factors such as products, competition and customers behavior. The work of [9] proposed formulating several economically incentivised applications (e.g., *potential customers identification*, *product feature promotion*, *product positioning*) as optimization problems taking a data mining perspective. In the context of database research, DADA [10] was the first of a series of works in the field proposing various queries by capitalizing on the dominance relationships among products and customer preferences.

Several works [5, 11, 25, 18, 6, 3] focus on identifying the potential customers of a product. In order to provide an insight on the product against the competition, [13, 24, 23] address the problem of discovering and promoting the best product features. Another practical application is how to design new products, such that they will maximize the expected utility, a problem known as product positioning [20, 19, 21, 12, 15]. The utility function may incorporate various factors such as the number of expected product buyers [19, 21, 12, 15], the actual profit (price minus production cost) [10, 20, 21, 15] or the number and features of other competitive products [10, 12]. Other works [20, 21] seek profitable packages formed by combining individual products, e.g., a flight and a hotel room, such that the profit gain of the package is maximized.

With regards to the number of expected customers, one problem is how to best model user preferences. One way is to assume that a weight vector capturing the importance of different product features (attributes) has been determined for each customer, through a

preference learning process. Based on this assumption, each product is assigned a score by applying the weight vector known for the user. Then, the products that score higher are those that would be more attractive to the respective user. This is the approach taken in *top-k queries* [7]. [18] tackled the reverse problem of discovering the most attractive products by introducing reverse top-k queries.

However, the weight vector formulation is often too difficult to come up with in real life [17]. A more natural way to model preferences is by allowing users to directly specify their preferred product attribute values. Taking this approach, both products and user preferences can be represented as points in a multidimensional space. In such a scenario, different notions of user satisfaction have been proposed. One option is to allow users to specify the worst acceptable value for each dimension [15]; all products having better values from the specified ones are considered as satisfactory. An important limitation of such a formulation is that it cannot be used for 'subjective' types of attributes. Further, there is no metric of how relevant each product is w.r.t. the actual user preferences. Thus, another option is to measure product attractiveness based on how close the product attribute values are to the user-preferred ones. In order to find the k most attractive products for a customer $c \in C$ we can issue a kNN(c) query [16] on the product data set. However, in several real applications it could be hard to find an appropriate distance function, because different dimensions might have different weights which depend on the preferences of each user.

With the goal to overcome the limitations of top-k and kNN queries, skylines have been widely used for multi-criteria decision analysis and for preference queries. The skyline query returns the set of not dominated objects, corresponding to the Pareto optimal set, which will always include the top-1 result for any monotone preference function. [4] introduces skyline queries in databases, also presenting various external memory algorithms. In order to capture subjective attributes, the dynamic skyline [14] returns all products that are 'attractive' according to a user's preferences.

Viewing the problem from a manufacturer's perspective, [5] introduces the reverse skyline query, which returns all customers that would find a product as 'attractive', and proposes a branch-and-bound extension of the BBS algorithm [14] that reduces the search space. [11] improves upon [5] by providing tighter pruning rules based on midpoint skylines (Section 2.2) and presents algorithms for calculating reverse skylines on uncertain data. [25] proposes the BRS algorithm (Section 2.3), which exploits additional optimizations for precise data. [6] considers non-metric attribute domains and proposes non-indexed algorithms to efficiently calculate the influence set in that case, whereas [22] studies how to process reverse skylines energy-efficiently in a wireless sensor network.

In this work, we formulate customers identification by following a reverse skyline approach, where we consider both subjective dimensions and competition. We focus on providing a more efficient and progressive algorithm for single-point reverse skylines. Further, we extend our methods for multiple query points, with applications to the k-MAC query. The methods proposed in [12, 15] cannot be applied onto our setting, because they assume the same product dominance relationships holding for all users and that they can be calculated by executing only one skyline query. However, this is not true when each user has his preferred attribute values.

8. CONCLUSIONS

In this work, we studied two classes of queries involving customer preferences with important applications in market research. We first proposed the RSA algorithm for reverse skyline query evaluation. We then developed a batched extension of our RSA algorithm that significantly improves upon processing multiple queries individually, by grouping contiguous candidates, exploiting I/O commonalities and enabling shared processing, and applied this batched extension to solve the k-MAC query. Our experimental study on both real and synthetic data sets demonstrates that (*i*) RSA is significantly more efficient, scalable and progressive than the BRS algorithm for the reverse skyline problem, and (*ii*) that our proposed batched algorithm is the best choice for the k-MAC query.

Acknowledgments This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund. The authors would also like to thank Prof. Dimitris Papadias and Prof. Timos Sellis for their guidance and support.

9. REFERENCES

[1] http://randdataset.projects.postgresql.org.
[2] J. Bentley, K. Clarkson, and D. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. In *SODA*, 1990.
[3] T. Bernecker, T. Emrich, H.-P. Kriegel, N. Mamoulis, M. Renz, S. Zhang, and A. Züfle. Inverse queries for multidimensional spaces. In *SSTD*, 2011.
[4] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.
[5] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *VLDB*, 2007.
[6] P. Deshpande and D. P. Efficient reverse skyline retrieval with arbitrary non-metric similarity measures. In *EDBT*, 2011.
[7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.
[8] D. Hochbaum and A. Pathria. Analysis of the greedy approach in problems of maximum k-coverage. *NRL*, 45, 1998.
[9] J. M. Kleinberg, C. H. Papadimitriou, and P. Raghavan. A microeconomic view of data mining. *Journal of Data Mining and Knowledge Discovery*, 2(4), 1998.
[10] C. Li, B. C. Ooi, A. K. H. Tung, and S. Wang. Dada: a data cube for dominant relationship analysis. In *SIGMOD*, 2006.
[11] X. Lian and L. Chen. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *SIGMOD*, 2008.
[12] C.-Y. Lin, J.-L. Koh, and A. L. Chen. Determining k-most demanding products with maximum expected number of total customers. *TKDE*, 2012.
[13] M. Miah, G. Das, V. Hristidis, and H. Mannila. Standing out in a crowd: Selecting attributes for maximum visibility. In *ICDE*, 2008.
[14] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *TODS*, 30(1), 2005.
[15] Y. Peng, R. C.-W. Wong, and Q. Wan. Finding top-k preferable products. *TKDE*, 2012.
[16] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, 1995.
[17] Y. Tao, L. Ding, X. Lin, and J. Pei. Distance-based representative skyline. In *ICDE*, 2009.
[18] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *ICDE*, 2010.
[19] A. Vlachou, C. Doulkeridis, K. Nørvåg, and Y. Kotidis. Identifying the most influential data objects with reverse top-k queries. *PVLDB*, 3(1), 2010.
[20] Q. Wan, R. C.-W. Wong, I. F. Ilyas, M. T. Özsu, and Y. Peng. Creating competitive products. *PVLDB*, 2(1), 2009.
[21] Q. Wan, R. C.-W. Wong, and Y. Peng. Finding top-k profitable products. In *ICDE*, 2011.
[22] G. Wang, J. Xin, L. Chen, and Y. Liu. Energy-efficient reverse skyline query processing over wireless sensor networks. *TKDE*, 24(7), 2011.
[23] T. Wu, Y. Sun, C. Li, and J. Han. Region-based online promotion analysis. In *EDBT*, 2010.
[24] T. Wu, D. Xin, Q. Mei, and J. Han. Promotion analysis in multi-dimensional space. *PVLDB*, 2(1), 2009.
[25] X. Wu, Y. Tao, R. C.-W. Wong, L. Ding, and J. X. Yu. Finding the influence set through skylines. In *EDBT*, 2009.

Scaling Out Big Data Missing Value Imputations

(Pythia vs. Godzilla)

Christos Anagnostopoulos
School of Computing Science
University of Glasgow, G12 8QQ, Glasgow, UK
christos.anagnostopoulos@glasgow.ac.uk

Peter Triantafillou
School of Computing Science
University of Glasgow, G12 8QQ, Glasgow, UK
peter.triantafillou@glasgow.ac.uk

ABSTRACT

Solving the missing-value (MV) problem with small estimation errors in big data environments is a notoriously resource-demanding task. As datasets and their user community continuously grow, the problem can only be exacerbated. Assume that it is possible to have a single machine ('Godzilla'), which can store the massive dataset and support an ever-growing community submitting *MV imputation* requests. Is it possible to replace Godzilla by employing a large number of cohort machines so that imputations can be performed much faster, engaging cohorts in parallel, each of which accesses much smaller partitions of the original dataset? If so, it would be preferable for obvious performance reasons to access only a subset of all cohorts per imputation. In this case, can we decide swiftly which is the desired subset of cohorts to engage per imputation? But efficiency and scalability is just one key concern! Is it possible to do the above while ensuring comparable or even better than Godzilla's imputation estimation errors? In this paper we derive answers to these fundamentals questions and develop principled methods and a framework which offer large performance speed-ups and better, or comparable, errors to that of Godzilla, independently of which missing-value imputation algorithm is used. Our contributions involve Pythia, a framework and algorithms for providing the answers to the above questions and for engaging the appropriate subset of cohorts per MV imputation request. Pythia functionality rests on two pillars: (i) dataset (partition) signatures, one per cohort, and (ii) similarity notions and algorithms, which can identify the appropriate subset of cohorts to engage. Comprehensive experimentation with real and synthetic datasets showcase our efficiency, scalability, and accuracy claims.

Categories and Subject Descriptors: H. Information Systems; I.5.3 Clustering.

Keywords: Big data; Missing value; Clustering.

1. INTRODUCTION

Data quality is a major concern in big data processing and knowledge management systems. One relevant problem in data quality is the presence of missing values (MVs). The MV problem should be carefully addressed, otherwise bias might be introduced into the induced knowledge. Common solutions to the MV problem either fill-in the MVs (*imputation*) or ignore / exclude them. Imputation entails a *MV substitution algorithm* (MVA) that replaces MVs in a dataset with some plausible values. Imputed data can be treated as reliable as the observed data, but they are as good estimations as the assumptions used to create them.

On the one hand, most computational intelligence and machine learning (ML) techniques (such as neural networks and support vector machines) fail if one or more inputs contains MVs and thus cannot be used for decision-making purposes [1]. Furthermore, the choice of different MVAs affects the performance of ML techniques that are subsequently used with imputed data [2]. On the other hand, the MV problem abounds: it can be found, for instance, in results from medical experimentation and chemical analysis, in datasets from domains such as meteorology and microarray gene monitoring technology [4], and in survey databases [5]. MVs can occur e.g., due to wireless sensor faults, not reacting experiments, or participants skipping survey questions. Industrial and research databases include MVs [6], e.g., maintenance databases have up to 50% of their entries missing [7]. Patient records in medical databases lack some values; interestingly, a database of patients with cystic fibrosis missing more than 60% of its entries was analyzed in [8]. Moreover, gene expression microarray data sets contain MVs, making the need for robust MVAs apparent, since algorithms for gene expression analysis require complete gene array data [9].

Motivations. Given the significance of MVAs, three notes are in order: Firstly, MVAs which can ensure low estimation errors are computationally expensive and typically their performance is largely dependent on dataset sizes. Secondly, nowadays, datasets can be massive. Even worse, existing datasets grow significantly with time; it is not surprising that most MVAs in the literature are typically tested over small-to medium sized datasets. Lastly, as if the scalability limitations imposed by dataset sizes were not enough, in many applications the user community (e.g., in shared scientific datasets in data centers accessed by scientists from all over the world) can be very large and thus the MV imputation input arrival rates can become high as well. These facts pose a scalability nightmare.

The scalability gospel (as established by the seminal work from Google researchers producing the Map-Reduce (MR) [10] data-access paradigm and systems such as the Google

KDD'14, August 24–27, 2014, New York, NY, USA.

http://dx.doi.org/10.1145/2623330.2623615.

File System [11]) rests on the notion of *scaling out*: that is, (i) employ a large number of commodity (off-the-shelf and thus inexpensive) machines, each storing a much smaller partition of the original dataset, and (ii) access them in parallel.

However, MR is not a panacea, for two reasons. First, not all complex problems are 'embarrassingly parallelizable' and amenable to MR techniques. In particular, there exist sophisticated MVAs ensuring small errors, which are not MR-able [12]. Second, in the context of MVAs, even if they were 'embarrassingly parallelizable', not all partitions may be relevant. It may very well be the case that a number of the machines hold data that cannot help (or even hurt) in the MV imputation process. And, obviously, engaging only a fraction of all machines will introduce large benefits: First with respect to performance. MV imputation will be shorter, as these times typically depend on the worst performing machine and with increasing machine numbers the probability of a mall-performing machine increases. Further, overall MV imputation throughput will be higher, as each imputation will be taxing fewer overall system resources (processors, communication bandwidth and disks). Second, with respect to MV estimation errors. In fact, as we shall formally show later, engaging all machines and their dataset partitions may actually introduce large additional MV estimation errors.

Goals. In this work, we will consider a stream of MV inputs (or inputs), i.e., multi-dimensional vectors with some MVs in certain dimensions, arriving at a data system. Typically, the system is presented with a batch of data items with MVs, which must be added to the system after MVs have been estimated. There are two system alternatives. The first is based on employing a single machine which stores the whole of the dataset. We affectionately call this machine *Godzilla*. Godzilla can employ any MVA to perform the MV imputations. As motivated earlier, this approach suffers from several disadvantages. The second alternative employs a (potentially large) number of machines, referred to as *cohorts*, each storing a partition of Godzilla's dataset. Imputation execution engages cohorts in parallel, whereby each cohort runs an MVA on a much smaller local dataset. This can introduce dramatic performance improvements. As an illustration, assuming, say, 50 cohorts and an MVA operating on a dataset of size n with asymptotic complexity $O(n^2)$ (or $O(n^3)$; [3], [4]) a scale-out execution is expected to speedup input processing by a factor of $50^2 = 2,500$ (or $50^3 = 125,000$) as such MVA runs in parallel on a dataset of size $\frac{1}{50}n$. Moreover, this alternative affords the possibility of accessing only a subset of all cohorts for a given input. We will not make any restricting assumptions as to specific characteristics of this system or the method for partitioning the dataset.

The formidable challenges here entail: (i) for data accuracy (estimation-error) reasons, we should ensure that the subset of cohorts contacted achieve similar, if not smaller estimation errors, compared to the errors that Godzilla would yield; (ii) *swiftly* determine the subset of cohorts to engage per imputation, achieving large efficiency/scalability gains.

Contributions. To our knowledge, this is the first study on scaling out MV imputations. We shall derive fundamental knowledge regarding meeting the estimation error and performance goals outlined above. Armed with this knowledge, we shall propose a novel, principally derived framework, *Pythia*, which offers large performance speed-ups and better, or comparable, errors to that of Godzilla given a stream of MV inputs. Pythia's salient contribution is that, given an input (of imputation requests), Pythia is able to predict and engage the appropriate subset of cohorts to employ per imputation. Pythia's prediction process relies on (i) the concept of per cohort-dataset *signature*, which derives from the (local) dataset of a cohort and (ii) novel similarity notions and algorithms which, based on each imputation request and cohort signatures, can determine the best subset of cohorts to engage. Finally, we will provide comprehensive experimental evidence substantiating and showcasing Pythia's accuracy and performance, using a variety of metrics and real and synthetic datasets.

The paper is structured as follows: Section 2 reports on background and discusses related work, while Section 3 introduces the problem fundamentals of scaling out the MV imputations. In Section 4 and Section 5 we introduce the Pythia framework and propose two schemes. In Section 6 we evaluate our framework and Section 7 concludes the paper.

2. BACKGROUND & RELATED WORK

2.1 Missing data

Assume a data set $\mathcal{X}$ of d-dimensional data points with some MVs on a certain dimension X_i. Data on X_i are said to be *missing completely at random* (MCAR) if the probability of MV on X_i, q, is unrelated to the value of X_i itself or to the values of any other dimensions. If data are MCAR, a reduced sample of $\mathcal{X}$ will be a random sub-sample of $\mathcal{X}$; MCAR assumes that the distributions of MVs and complete data are the same. Data on X_i are said to be *missing at random* (MAR) if q depends on the observed data, but does not depend on the MV itself. In MAR, the dimension associated with MVs has a relation to other dimensions, i.e., MVs can be estimated by using the complete data of other dimensions. It is impossible to test whether the MAR condition is satisfied for $\mathcal{X}$ because, since the (actual) values of missing data are not known, we cannot compare the values of those with and without missing data to see if they differ systematically on that X_i. Data on X_i are *missing not at random* (MNAR) if q depends on the MVs and, thus, missing data cannot be estimated by using the existing dimensions; MNAR is rarely applicable in practice.

2.2 Related work

Missing data hinder the application of many statistical analysis and ML techniques available in off-the-shelf software. To analyze $\mathcal{X}$ with MVs, certain MVAs have been proposed [13]. The simplest method is discarding the data points with MVs or removing the corresponding dimensions. Both removals of such points and dimensions result in decreasing the information content of $\mathcal{X}$ and are applicable only when (i) $\mathcal{X}$ contains a small amount of MVs, and (ii) the analysis of the remaining complete points will not be biased by the removal. There are many MVAs varying from naïve methods, e.g., mean imputation, to some more robust methods based on relationships among dimensions. In the *dummy variable adjustment*, MVs are set to some arbitrary value. The *mean / mode imputation* replaces MVs of a dimension by the sample mean / mode of all observed values of that dimension. In *hot deck* MVA [14], a MV is filled in with a value from an estimated distribution w.r.t. $\mathcal{X}$. In the K-nearest neighbors MVA [15], the MVs of a point are imputed considering the K most similar (observed) points

from $\mathcal{X}$. The regression- and likelihood-based MVAs are introduced in [16]. In *regression-based imputation* [17], the MVs of a point are estimated by regression of the dimensions corresponding to MVs on the dimensions associated to the observed values of that point. This approach argues that dimensions have relationships among themselves; if no relationships exist among dimensions in $\mathcal{X}$ and the dimensions corresponding to MVs, such MVA will not be precise for imputation. *Likelihood-based imputation* [16] is based on parameter estimation in the presence of MVs, i.e., $\mathcal{X}$'s parameters are estimated by maximum likelihood or maximum a posteriori procedures relying on variants of the Expectation-Maximization algorithm. The *multiple imputation* MVA [18], instead of filling in a single value for each MV, replaces each MV with a set of plausible values that represent the uncertainty about the actual value to impute. These multiply-imputed datasets are then analyzed by using standard procedures for complete data and combining the results from these analyses. In case of MVs in time series, the models in [19] (using dynamic Bayesian networks), [20] (using matrix completion), and [21] (using Gaussian mixtures clustering) recover MVs in motion capture sequences, vital signs, and micro-array gene expression streams, respectively. Furthermore, ML-based MVAs, e.g., decision-trees and rule-based methods, generate a model from $\mathcal{X}$ that contain MVs, which is used to perform classification that imputes the MVs (see [2] and the references therein). Finally, the imputation framework [6] applies most existing MVAs (base methods) to improve their accuracy of imputation while preserving the asymptotic computational complexity of the base methods. The interested reader could also refer to [6], [9] and [22] (and the references therein) for a comprehensive survey of the most recent MVAs.

3. PROBLEM ANALYSIS & FUNDAMENTALS

3.1 Definitions & Notations

Definition 1. Given a set $\mathcal{X}$ of d-dimensional data points, $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_{|\mathcal{X}|}\}$, for each $\mathbf{x}_i$ we define $\mathbf{w}_i = [w_{ik}]^\top$ with $w_{ik} = 0$ whenever $\mathbf{x}_i$'s k-th dimensional value is missing; otherwise $w_{ik} = 1$. We express $\mathbf{x}_i$ as $(\mathbf{z}_i, \mathbf{z}_i^{\mathbf{m}})$, where $\mathbf{z}_i \in \mathbb{R}^n$ denotes observed values and $\mathbf{z}_i^{\mathbf{m}} \in \mathbb{R}^{(d-n)}$ denotes MVs, with $n = \sum_{k=1}^{d} w_{ik}$.

Definition 2. Given a finite integer $m > 0$, $\mathcal{X}_i$ is a partition of $\mathcal{X}$ such that $\mathcal{X} \equiv \cup_{i=1}^{m} \mathcal{X}_i$ and $\mathcal{X}_i \neq \mathcal{X}_j, i \neq j$. S_i denotes the machine (*cohort*), which maintains $\mathcal{X}_i$, performs a MVA over $\mathcal{X}_i$, and is indexed by i, $i = 1, \ldots, m$. $\mathcal{S} = \{S_i\}_{i=1}^{m}$ is the set of all cohorts. The (imaginary) *Godzilla* S_0 assembles all $\mathcal{X}_i$ and is capable of performing a MVA over $\mathcal{X}$.

Definition 3. A single MV input on MVA is $\mathbf{i} = (\mathbf{x}, \mathbf{w})$ and output is $\hat{\mathbf{x}}$ expressed by $(\mathbf{z}, \hat{\mathbf{z}}^{\mathbf{m}})$. $\hat{\mathbf{x}} \in \mathbb{R}^d$ is referred to as *estimate* containing $\hat{\mathbf{z}}^{\mathbf{m}} \in \mathbb{R}^{(d-n)}$ of imputed MVs by MVA. If $\mathbf{x}_a$ is the actual vector, the absolute reconstruction error is $e = \| \hat{\mathbf{x}} - \mathbf{x}_a \|$; $\| \mathbf{x} \|$ denotes the Euclidean norm.

3.2 MVAs in our framework

As our contributions are independent of any particular MVA, we overview two popular and representative MVAs as would be used in our framework. To exemplify our framework and methods, we employ the weighted K-nearest neighbors (KNN) [15] and sequential multivariate regression imputation method (REG) [17]. These MVAs are widely used for multivariate imputation in many scientific areas.

3.2.1 Weighted K-nearest neighbors imputation

KNN is widely used [22] since it has many attractive characteristics: it is a non-parametric method, which does not require the creation of a predictive model for each dimension with MV and takes into account the correlation structure of the data. KNN is based on the assumption that points close in distance are potentially similar. For given input $(\mathbf{x}_i, \mathbf{w}_i)$ with $\mathbf{x}_i = (\mathbf{z}_i, \mathbf{z}_i^{\mathbf{m}})$, KNN calculates a weighted Euclidean distance D_{ij} between $\mathbf{x}_i$ and $\mathbf{x}_j \in \mathcal{X}$ such that

$$D_{ij} = \left(\frac{\sum_{k=1}^{d} w_{ik} w_{jk} (x_{ik} - x_{jk})^2}{\sum_{k=1}^{d} w_{ik} w_{jk}} \right)^{1/2}.$$

The MV of the k-th dimension of $\mathbf{x}_i$ (i.e., $z_{ik}^{\mathbf{m}}$ of $\mathbf{z}_i^{\mathbf{m}}$) is estimated by the weighted average of non-MVs of the K most similar $\mathbf{x}_j$ to $\mathbf{x}_i$, i.e., $\hat{z}_{ik}^{\mathbf{m}} = \sum_{j=1}^{K} \frac{D_{ij}^{-1}}{\sum_{v=1}^{K} D_{iv}^{-1}} x_{jk}$. KNN is typically used with K=10,15,20; theses values have been favored in previous studies [22], [23]. (In our experiments we will use K=10).

3.2.2 Sequential multivariate regression imputation

REG estimates the MVs by fitting a sequence of regression models and drawing values from the corresponding predictive distributions. Let $Y_1, \ldots, Y_{d-n}$ denote $d-n$ (dependent) variables with MVs, sorted in ascending order to the number of MVs and $\mathbf{X} = [X_1, \ldots, X_n]^\top$ denote n (predictor) variables with no MVs. REG consists of c rounds. In round 1, step 1, we regress the variable with the fewest number of MVs, Y_1, on $\mathbf{X}$ imputing the MVs under the appropriate regression model; e.g., if Y_1 is continuous, categorical, or binary then ordinary least squares, generalized logit, or logistic linear regression is applied, respectively. In step 2, after estimating the regression coefficients β of the model from step 1, we use the estimated $\hat{\beta}$ to impute the MVs of Y_1. In step 3, we update $\mathbf{X}$ by appending Y_1 and continue to variable, say Y_2, with the next fewest MVs and repeat the process using updated $\mathbf{X}$ as predictors until all the variables have been imputed. That is, Y_1 is regressed on $\mathbf{U} = \mathbf{X}$; Y_2 is regressed on $\mathbf{U} = (\mathbf{X}, Y_1)$, where Y_1 has imputed MVs; Y_3 is regressed on $\mathbf{U} = (\mathbf{X}, Y_1, Y_2)$, where Y_1 and Y_2 have imputed MVs, and so on. Steps 1 to 3 are then repeated in rounds 2 through c, modifying the predictors set to include all Ys except the one used as the dependent variable. Hence, regress Y_1 on $\mathbf{X}$ and $Y_2, \ldots, Y_{d-n}$; regress Y_2 on $\mathbf{X}$ and $Y_1, Y_3, \ldots, Y_{d-n}$, and so on. Repeated cycles continue for c rounds, or until stable imputed MVs occur.

3.3 On Cohort vs. Godzilla errors

We consider a discrete time domain $t \in \mathbb{T}$ and at instance $t = 1, 2, \ldots$, we are given input $\mathbf{i}[t]$. Assume that Godzilla S_0 exists and is capable of invoking a certain MVA for $\mathbf{i}[t]$. At first thought, one could claim that, since Godzilla has global knowledge (i.e., the union of all $\mathcal{X}_i$), the corresponding estimate $\hat{\mathbf{x}}_G[t]$ would be *better* (in terms of reconstruction error $e_G[t]$) than $\hat{\mathbf{x}}_i[t]$ of each S_i (with error $e_i[t]$). However, this does not always hold true. It depends on the probability density function (pdf) of $\{\mathcal{X}_i\}$ and the (possibly unknown) pdf of $\mathbf{z}[t]$, $\mathbf{z}^{\mathbf{m}}[t]$, and $\mathbf{w}[t]$.

THEOREM 1. *Let e_G and e_i denote the estimate error of Godzilla S_0 and cohort S_i. It is not always true that $e_G < e_i, \forall S_i \in \mathcal{S}$.*

PROOF. To prove Theorem 1, suppose its converse were true. Then it suffices to show counterexamples. Consider the mean imputation (MEAN) and the KNN. Consider that points in $\mathcal{X}_i$ are normally distributed, $\mathcal{N}(\mu_i, \sigma_i^2)$, with mean μ_i and variance σ_i^2 and $|\mu_i - \mu_j| >> 0, i \neq j$. Evidently, S_0's data set $\mathcal{X} = \cup_{i=1}^{m} \mathcal{X}_i$ follows the mixture $\mathcal{N}(\mu, \sigma^2)$ with $\mu = \sum_{i=1}^{m} a_i \mu_i$, $\sigma^2 = \sum_{i=1}^{m} a_i((\mu_i - \mu)^2 + \sigma_i^2)$; $a_i > 0, \sum_{i=1}^{m} a_i = 1$. If we were told that all (both observed $\mathbf{z}$ and unobserved $\mathbf{z^m}$) inputs followed $\mathcal{N}(\mu_j, \sigma_j^2)$ for some $j, 1 \leq j \leq m$ then we should have engaged only S_j thus yielding $e_j < e_G$ in case of MEAN, and $e_j = e_G$ in case of KNN (for $K << |\mathcal{X}_j|$) and avoiding engaging all S_i. □

Furthermore, consider that all $\mathcal{X}_i$ follow exactly the same distribution; consequently, S_0's $\mathcal{X}$ follows the same distribution. Then, regardless of any knowledge on the pdfs of inputs, we could randomly select one cohort from $\mathcal{S}$, thus, yielding $e_i = e_G, \forall S_i \in \mathcal{S}$, and avoiding engaging all cohorts.

Example 1: Consider $m = 3$ cohorts S_1, S_2, S_3 with 2D datasets $\mathcal{X}_i$, corresponding joint pdfs f_1, f_2, f_3 and a Godzilla S_0 with $\mathcal{X} = \cup_{i=1}^{3} \mathcal{X}_i$ whose joint pdf f_G is shown in Fig. 1(a). Assume REG, KNN, and MEAN MVAs. We are given a stream of 10^4 inputs $\mathbf{i}[1], \ldots, \mathbf{i}[t]$ and assume that we *know* the pdf of each $\mathbf{i}[t]$, i.e., its observed and MVs are known to be produced either by f_1, f_2, or f_3. For each $\mathbf{i}[t]$, we invoke a MVA (a) on S_0 and obtain $e_G[t]$, (b) only on the cohort S_j with the same pdf f_j as that of the input and obtain $e_j[t]$, and (c) on all cohorts, aggregate their estimates by taking their average and obtain $e_{all}[t]$. Fig. 1(b) shows the root-mean-square error (RMSE) e_G, e_j, and e_{all} for all MVAs. We observe that the knowledge of the pdf of each input results to a significantly lower error e_j, because we engage only the cohort S_j corresponding to the same pdf as that of the input. Godzilla produces a relatively high e_G (for all MVAs) with high computational cost due to processing high volumes of data. Moreover, the parallel execution of MVAs over all cohorts for each input produces a high e_{all}. Unfortunately, the pdf of an incoming input is not known, especially, the pdf of the MVs is unknown since they are never observed. Moreover, we can achieve high parallelism with concurrently engaging all cohorts but, we also obtain high error, because there might be a subset of cohorts that *adversely* contribute to the aggregated estimate, e.g., due to the fact that the corresponding pdfs of their data sets are different from those of the inputs (see Example 2). Note, however, that in the case of MEAN, $e_G = e_{all}$.

3.4 On computing good cohort subsets

Here we show: (i) that computing the best cohorts subset is computationally hard, (ii) that even if an efficient heuristic can be found, it would not be desirable for our purpose since it would require communication with all cohorts, hence, another approach is needed, like our signature-based prediction approach and (iii) that as exemplified using our reference popular MVAs, it is highly beneficial to engage only a good cohort subset per imputation. The above showcases thus the traits and benefits of our approach.

In our framework, we utilize a node called Pythia that attempts to predict the best cohorts subset per input. Pythia receives input $\mathbf{i}[t] = (\mathbf{x}[t], \mathbf{w}[t])$ with $0 < n[t] = \sum_{k=1}^{d} w_k[t] < d$. In the remainder, the time index t is omitted for the sake of readability. Of course, Pythia can, trivially, engage all cohorts in parallel. Each cohort S_i locally produces an estimate $\hat{\mathbf{x}}_i$ (through MVA invocation) and provides it to Pythia. Then, Pythia takes their average value $\hat{\mathbf{x}} = \frac{1}{m} \sum_{i=1}^{m} \hat{\mathbf{x}}_i$. Let us denote such method as the *All Cohorts Method*, notated by ACM, so to differentiate it from Pythia's sophisticated methods. ACM implies that all cohorts are equal candidates and available for providing an estimate. It would have been preferable if Pythia could engage a subset $\mathcal{S}' \subset \mathcal{S}$ of cohorts whose average estimate $\hat{\mathbf{x}}' = \frac{1}{|\mathcal{S}'|} \sum_{S_i \in \mathcal{S}'} \hat{\mathbf{x}}_i$ would be equal to $\hat{\mathbf{x}}$, or more interestingly, if Pythia could engage the minimum subset of cohorts whose average estimate is close to $\hat{\mathbf{x}}$ for each input.

Determining the minimum cohorts subset whose aggregate estimate is close to $\hat{\mathbf{x}}$ calls to mind the Subset Sum Problem (SSP) [24]: Consider a pair $(\mathcal{I}, s)$, where $\mathcal{I}$ is a set of $m > 0$ positive integers and s is a positive integer. SSP asks for a subset of $\mathcal{I}$ whose sum is closest to, but not greater than, s. SSP is NP-hard [24]. Consider now the following problem, referred to as Minimum Subset Average Problem (MSAP).

Problem 1. (MSAP) Given $(\mathcal{I}, s)$, find the minimum subset $\mathcal{I}'$ with average s' subject to $\lfloor s' \rfloor = s$ or $\lceil s' \rceil = s$ (C1).

THEOREM 2. *MSAP is NP-hard.*

PROOF. If there is a polynomial-time algorithm for MSAP, then a polynomial-time algorithm can be developed for SSP. Assume there exists a polynomial algorithm $A(\mathcal{I}, s)$ that solves MSAP, i.e., $A(\mathcal{I}, s)$ finds in polynomial time the minimum subset $\mathcal{I}'$ subject to constraint C1 in Problem 1. Then, $A(\mathcal{I}, s)$ can be used to solve SSP with $(\mathcal{I}, ms)$, $m = |\mathcal{I}|$. In general, any solution $B(\mathcal{I}, s)$ of SSP with $(\mathcal{I}, s)$ can be formulated as Algorithm 1. If the complexity of $A(\mathcal{I}, s)$ is a polynomial $\mathcal{Q}(m)$ then the complexity of $B(\mathcal{I}, s)$ is $O(m\mathcal{Q}(m))$. But, this implies that there is a polynomial-time algorithm for SSP. Hence, no polynomial-time algorithm exists for MSAP. □

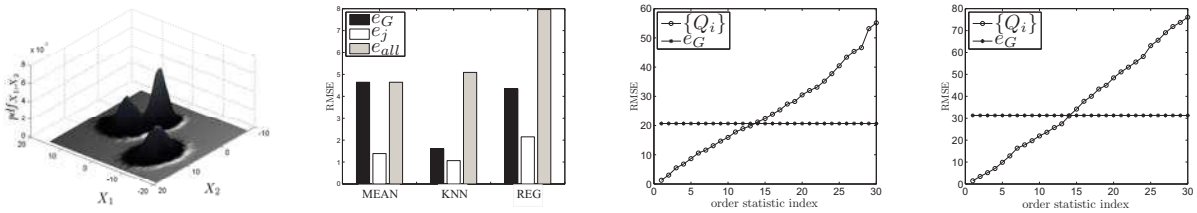
ALGORITHM 1: $B(\mathcal{I}, s)$

```
Input: I, s
Output: I'
for 1 ≤ k ≤ |I| do
    call A(I, s/k);
    If a subset I' of I with k elements is found, whose
    elements have an average k' such that ⌊k'⌋ = s/k or
    ⌈k'⌉ = s/k Then return I'
end
```

THEOREM 3. *Given input $\mathbf{i}$, the problem of finding the minimum subset $\mathcal{S}' \subset \mathcal{S}$ of cohorts, whose average estimate $\hat{\mathbf{x}}'$ gives the same reconstruction error as $\hat{\mathbf{x}}$ is NP-hard.*

PROOF. Let $e = \| \hat{\mathbf{x}} - \mathbf{x}_a \|$ and $e' = \| \hat{\mathbf{x}}' - \mathbf{x}_a \|$. In order to show that the problem of finding the minimum subset $\mathcal{S}'$ with $e' = e$ is NP-hard, it suffices to show that finding the minimum subset $\mathcal{S}' \subset \mathcal{S}$ of cohorts such that $\| \hat{\mathbf{x}}' \| = \| \hat{\mathbf{x}} \|$ subject to C1 is NP-hard. Consider the set $\mathcal{I}^0 = \{\lfloor \| \hat{\mathbf{x}}_i \| \rfloor\}_{i=1}^{m}$, and $\mathcal{I}^1 = \{\lceil \| \hat{\mathbf{x}}_i \| \rceil\}_{i=1}^{m}$, $\| \hat{\mathbf{x}}_i \| > 0, \forall i$. Since MSAP, which deals with integers is NP-hard from Theorem 2, MSAP with $(\mathcal{I}^0, \lfloor \| \hat{\mathbf{x}} \| \rfloor)$ and $(\mathcal{I}^1, \lceil \| \hat{\mathbf{x}} \| \rceil)$ is also NP-hard. □

SSP and MSAP are NP-hard, however, one is often satisfied with an approximate, sub-optimal solution, i.e., in polynomial time; see [25] for SSP. Nevertheless, even if Pythia



(a) Joint pdf $f_G(X_1, X_2)$ (b) RMSE using MEAN, KNN, REG (c) RMSE using KNN (d) RMSE using REG

Figure 1: (a) Joint pdf; (b) RMSE e_G, e_j, e_{all} using MEAN, KNN and REG for $m = 3$ in Example 1; (c-d) RMSE of Godzilla and order statistics Q_i of ACM using KNN and REG for $m = 30$ in Example 2.

were able to use such heuristic to find the minimum set $\mathcal{S}'$ for given input (let m be small) then this would still not be preferable given our goals. That is because, in order to obtain $\mathcal{S}'$ for a given input, Pythia would *firstly* have to engage all cohorts and consequently, based on their estimates, produce $\mathcal{S}'$. What we want is for Pythia to guess/predict the most appropriate $\mathcal{S}'$, which gives the same or, hopefully, smaller reconstruction error than that of $\mathcal{S}$ *without having to access all cohorts*! For instance, this guess can be interpreted as follows: cohort $S_i \in \mathcal{S}$ might consider $\mathbf{z}$ (of input $\mathbf{i}$) as an observation which is deemed *unlikely* w.r.t. $\mathcal{X}_i$. Based on the fact that a MVA highly depends on $\mathcal{X}_i$, S_i will probably provide a bad estimate for $\mathbf{i}$ (w.r.t. e_i). Were Pythia capable of predicting the *unsuitability* of S_i providing a good estimate *before* engaging S_i then Pythia could have excluded S_i from $\mathcal{S}'$.

The task of predicting $\mathcal{S}'$ per input involves the following issues: (a) the joint pdf of the MVs is evidently unknown since the actual values of $\mathbf{z^m}$ are not observed; (b) it is not feasible to identify the joint pdf that generates $\mathbf{z}$, since we have only one sample from this at a time; (c) it is not suitable to assume that $\mathbf{z}$ is produced by a certain pdf at time t, which remains also the same for subsequent $\mathbf{z}[\tau], \tau > t$. This is getting more difficult when dealing with non-stationary distributions of $\mathbf{z}$ and $\mathbf{w}$, which is not a rare situation.

Example 2: Consider $m = 30$ cohorts. We are given a stream of 10^4 inputs where the joint pdf of each input is unknown. For each input, we invoke a MVA (KNN and REG) on Godzilla and on all cohorts in parallel, and aggregate their estimates (ACM). For each input, we obtain the order statistics $Q_1 = \min_i\{e_i\}, \ldots, Q_{30} = \max_i\{e_i\}$ of the corresponding errors of all cohorts and plot their average values in Fig. 1(c-d); the e_G is shown for comparison. We can observe that more than 40% of cohorts provide lower error to that of Godzilla for KNN and REG. This indicates that it is of high importance to predict such subset of cohorts for each input while knowing neither the pdfs of the cohorts' sets nor the pdf of each input. Note that ACM in this case produces a higher average error than even Godzilla. Furthermore, we observe that for each input there is an *ideal* cohort that gives the minimum error; note that $\bar{Q}_1$ is 93% / 95% smaller than e_G for KNN / REG. An *ideal* Pythia has to predict $\mathcal{S}'$ hopefully including the ideal cohort and/or those S_i with $e_i < e_G$ for each input. We now formulate our problems.

Problem 2. Determine what *information* each $S_i \in \mathcal{S}$ apriori must convey to Pythia in order to predict whether S_i is suitable for providing a (local) good estimate $\hat{\mathbf{x}}_i$ given an input, i.e., whether S_i should be a member of $\mathcal{S}'$. This information is referred to as the *signature* P_i of $\mathcal{X}_i$.

Problem 3. Determine how signatures $\{P_i\}_{i=1}^m$ are updated for each input.

4. THE PYTHIA FRAMEWORK

Pythia aims to solve the above problems. Predicting $\mathcal{S}'$ for each input, based on per-cohort signatures, avoids the fundamental problems of NP-hardness of exact solutions and of the need to on-the-fly engage all cohorts for approximate heuristic solutions.

Each cohort S_i constructs a signature P_i from $\mathcal{X}_i$. P_i reflects the current structure of data points in $\mathcal{X}_i$. The idea behind a signature is that S_i is engaged for a given $\mathbf{i}$ once $\mathbf{x}$ can be 'explained' through P_i. S_i provides its (locally) created P_i to Pythia, which stores all signatures forming $\mathcal{P} = \{P_i\}_{i=1}^m$. Figure 2(a) pictorially depicts the framework's operation. The operation of the framework is as follows: Given $\mathbf{i}$,

1. Pythia predicts $\mathcal{S}' \subseteq \mathcal{S}$ w.r.t. $\mathcal{P}$
2. Pythia then engages only the cohorts from S' sending $\mathbf{i}$ to them.
3. Each $S_i \in \mathcal{S}'$
   (a) invokes a MVA and
   (b) provides its estimate $\hat{\mathbf{x}}_i$ to Pythia.
4. Pythia constructs the aggregate estimate $\hat{\mathbf{x}}$ that is sent to cohorts from $\mathcal{S}'$.
5. Each $S_i \in \mathcal{S}'$ can exploit $\hat{\mathbf{x}}$ for updating its P_i.
6. Pythia uses $\hat{\mathbf{x}}$ for updating $\mathcal{P}$.

4.1 Signatures

In this work, P_i refers to a clustering structure over $\mathcal{X}_i$ providing a set of representative points (clusters) $\mathcal{C}_i$. Each cohort $S_i \in \mathcal{S}$ employs the Adaptive Resonance Theory (ART) [26], an unsupervised learning model from the competitive learning paradigm, in order to locally construct P_i over $\mathcal{X}_i$. In ART, whose algorithm is shown as Algorithm 2, each $\mathbf{x}_k \in \mathcal{X}_i$ is processed by finding the nearest cluster $\mathbf{c}^* \in \mathbb{R}^d$ to $\mathbf{x}_k$, i.e., $\mathbf{c}^* = \arg\min_{\mathbf{c}\in\mathcal{C}_i} \parallel \mathbf{c} - \mathbf{x}_k \parallel$, where $\mathcal{C}_i$ is the set of clusters. Then, it is allowed $\mathbf{x}_k$ to modify/update $\mathbf{c}^*$ only if $\mathbf{c}^*$ is sufficiently close to $\mathbf{x}_k$ ($\mathbf{c}^*$ is said to 'resonate' with $\mathbf{x}_k$) i.e., if $\parallel \mathbf{c}^* - \mathbf{x}_k \parallel \leq \rho_i$ for some *vigilance* $\rho_i > 0$. In this case, $\mathbf{c}^*$ is updated through the rule $\mathbf{c}^* \leftarrow \mathbf{c}^* + \eta_i(\mathbf{x}_k - \mathbf{c}^*)$, where $\eta_i \in (0, 1)$ is a learning rate, which gradually decreases. Otherwise, i.e., $\parallel \mathbf{c}^* - \mathbf{x}_k \parallel > \rho_i$, a new cluster $\mathbf{c}$ is formed handling $\mathbf{x}_k$ such that $\mathbf{c} = \mathbf{x}_k$ and $\mathcal{C}_i \leftarrow \mathcal{C}_i \cup \{\mathbf{c}\}$.

Definition 4. Cohort S_i's signature P_i over $\mathcal{X}_i$ is the triple

$$P_i = \langle \mathcal{C}_i, \rho_i, \eta_i \rangle. \quad (1)$$

ALGORITHM 2: ART algorithm at cohort S_i

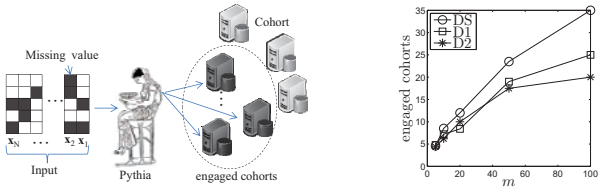
Input: $\mathcal{X}_i, \eta_i, \rho_i$
Output: $\mathcal{C}_i$
$\mathcal{C}_i = \{\mathbf{x}_1\}$;
for $1 < k \leq |\mathcal{X}_i|$ **do**
  $b^* = \| \mathbf{c}^* - \mathbf{x}_k \| = \min_{\mathbf{c} \in \mathcal{C}_i} \| \mathbf{c} - \mathbf{x}_k \|$;
  if $b^* > \rho_i$ **then**
    $\mathcal{C}_i \leftarrow \mathcal{C}_i \cup \{\mathbf{x}_k\}$;
  else
    $\mathbf{c}^* \leftarrow \mathbf{c}^* + \eta_i(\mathbf{x}_k - \mathbf{c}^*)$;
  end
end

Definition 5. We say that $\mathbf{x}$ is a *member* of P_i, notated $\mathbf{x} \in P_i$, iff $\min_{\mathbf{c} \in \mathcal{C}_i} \| \mathbf{c} - \mathbf{x} \| \leq \rho_i$; otherwise, $\mathbf{x} \notin P_i$.

The statement '$\mathbf{x} \in P_i$' denotes that there is at least one $\mathbf{c} \in \mathcal{C}_i$ such that $\mathbf{x}$ is placed close to $\mathbf{c}$ with distance less than ρ_i, for instance, the closest cluster $\mathbf{c}^*$ to $\mathbf{x}$. The more clusters $\mathbf{c} \in \mathcal{C}_i$ satisfy the criterion $\| \mathbf{c} - \mathbf{x} \| \leq \rho_i$, the more appropriate $\mathcal{C}_i$ is for $\mathbf{x}$. In this sense, if $\mathbf{x} \in P_i$ then $\mathbf{x}$ can be represented by at least one cluster from $\mathcal{X}_i$. Based on this intuition, if $\mathbf{x} \in P_i$, cohort S_i provides a rather good estimate for some missing parts of $\mathbf{x}$ compared to a cohort S_j associated with a P_j for which it holds true that $\mathbf{x} \notin P_j$. The latter case indicates that no cluster from $\mathcal{C}_j$ can be a representative point for $\mathbf{x}$.

Since ρ_i represents a threshold of similarity between points and clusters, thus, guiding ART in determining when a new cluster should be formed, it should depend on $\mathcal{X}_i$. In order to give a physical meaning to ρ_i, it is expressed through a set of percentages $\alpha_k \in (0, 1)$ of the ranges between the lowest $x_k^{\min}$ and highest $x_k^{\max}$ values of each dimension k of points in $\mathcal{X}_i$, $k = 1, \ldots, d$. Let $\mathbf{r}_i = [(x_1^{\max} - x_1^{\min}), \ldots, (x_d^{\max} - x_d^{\min})]^\top$ and the diagonal $d \times d$ matrix $\mathbf{A}$ with $\mathbf{A}[k, k] = \alpha_k$. Then $\rho_i = \| \mathbf{A}\mathbf{r}_i \|$. High α_k values result to a low number of clusters and vice versa. Each S_i determines a ρ_i over $\mathcal{X}_i$, creates P_i through Algorithm 2, and sends P_i to Pythia.

Note: when dealing with mixed-type data points, e.g., consisting of categorical, binary, and continuous attributes, we can adopt appropriate distance metrics [27] for the distance between $\mathbf{x}_k$ and $\mathbf{x}_l$ instead of using the Euclidean distance $\| \mathbf{x}_k - \mathbf{x}_l \|$; this does not spoil the generality of signature creation.



(a) Pythia

(b) Number of engaged cohorts

Figure 2: (a) Inputs with MVs, Pythia and engaged cohorts; (b) Engaged cohorts against m (COE).

4.2 Cohort prediction schemes

Up to this point, we have shown how to use signatures as a guiding light to select appropriate cohorts for MV imputations. Now, our concern is twofold: MV imputations must be (i) low cost and (ii) high accuracy. Low cost (once signature processing is performed) refers to the communication cost between Pythia and cohorts and to the cost of running MVAs at cohorts. High accuracy refers to low RMSE. Therefore, we present algorithms with these in mind.

4.2.1 Cost-aware algorithm: Top-$\mathcal{K}$ Cohort scheme

For simplicity we present the top-1 (Best) Cohort (BC) scheme, i.e., $\mathcal{K} = 1$. Pythia is not involved in producing the (final) estimate $\hat{\mathbf{x}}$, instead, only one cohort (best cohort) is engaged for doing this locally. Pythia communicates only with the best cohort, which runs the MVA, thus, this optimizes our cost metric. Given $\mathbf{i}$, Pythia determines the best cohort $S^* \in \mathcal{S}$ with $P^* = \langle \mathcal{C}^*, \rho^*, \eta^* \rangle$ such that (**A1**) $\mathbf{c}^* = \arg\min_{\mathbf{c} \in \cup_{i=1}^m \mathcal{C}_i} \| \mathbf{c} - \mathbf{z} \|$ and $\mathbf{c}^* = \arg\min_{\mathbf{c} \in \mathcal{C}^*} \| \mathbf{c} - \mathbf{z} \|$, i.e., $\mathbf{c}^* \in \mathcal{C}^*$ is the closest cluster to $\mathbf{z}$ among all clusters from all signatures, and (**A2**) $\mathbf{z} \in P^*$. Note that $\mathbf{z} \in \mathbb{R}^n$ with $0 < n = \sum_{k=1}^d w_k < d$ provided that $\mathbf{x}$ contains $d - n$ MVs. In order to evaluate '$\mathbf{z} \in P^*$' Pythia calculates $\rho^{*(n)} \leq \rho^*$ associated with the n dimensions of $\mathbf{r}^*$ corresponding to the n non-MVs. Then, it checks if $\| \mathbf{c}^* - \mathbf{z} \| \leq \rho^{*(n)}$ dealing only with the n dimensions of $\mathbf{c}^*$. Pythia engages only S^*, which produces the final $\hat{\mathbf{x}}$. If there is no cohort that satisfies criteria A1 and A2, BC engages the cohort that satisfies only criterion A1. If $\mathcal{K} > 1$ one can repeat the above criteria for the top $\mathcal{K}$ cohorts ranked with the distance between the corresponding $\mathbf{c}_j^*$ and $\mathbf{z}$, $1 \leq j \leq \mathcal{K} < m$. In this case the final $\hat{\mathbf{x}}$ is produced by aggregating all $\hat{\mathbf{x}}_j$.

4.2.2 Accuracy-aware algorithm: Cohorts Outlier Elimination scheme

Cohorts Outlier Elimination (COE) trades off additional cost for improving our other metric, accuracy. Given $\mathbf{i}$, Pythia checks whether $\mathbf{z} \in P_i$. This is achieved once Pythia, for each cohort S_i, calculates $\rho_i^{(n)} \leq \rho_i$ associated with the n dimensions of $\mathbf{r}_i$ corresponding to the n non-MVs. If $\| \mathbf{c}_i^* - \mathbf{z} \| \leq \rho_i^{(n)}$ (dealing only with the n dimensions of $\mathbf{c}^*$) with $\mathbf{c}_i^* = \arg\min_{\mathbf{c} \in \mathcal{C}_i} \| \mathbf{c} - \mathbf{z} \|$ then $\mathcal{S}' \leftarrow \mathcal{S}' \cup \{S_i\}$. Once $\mathcal{S}'$ is determined with $\ell = |\mathcal{S}'| \leq |\mathcal{S}| = m$, Pythia engages only cohorts from $\mathcal{S}'$ and obtains their corresponding estimates $\hat{\mathbf{x}}_i, i = 1, \ldots, \ell$. The aggregate estimate $\hat{\mathbf{x}}$ determined by Pythia is

$$\hat{\mathbf{x}} = \sum_{S_i \in \mathcal{S}''} \hat{\mathbf{x}}_i b_i \ , \ b_i = \frac{\| \mathbf{z} - \mathbf{c}_i^* \|^{-1}}{\sum_{S_j \in \mathcal{S}''} \| \mathbf{z} - \mathbf{c}_j^* \|^{-1}}, \tag{2}$$

where b_i is the weight for estimate $\hat{\mathbf{x}}_i$ normalized by the sum of inverse distance from the closest cluster $\mathbf{c}_i^*$ to $\mathbf{z}$ from cohort $S_i \in \mathcal{S}''$. The set $\mathcal{S}'' \subseteq \mathcal{S}'$ contains cohorts $S_i \in \mathcal{S}'$ whose estimates are not considered *outliers* in $\mathcal{E} = \{\| \hat{\mathbf{x}}_1 \| , \ldots, \| \hat{\mathbf{x}}_\ell \|\}$. This is achieved by computing the statistic

$$u_{i,\mathcal{E}} = \frac{| \| \hat{\mathbf{x}}_i \| - median(\mathcal{E})|}{mad(\mathcal{E})} \tag{3}$$

for each $\| \hat{\mathbf{x}}_i \| \in \mathcal{E}$ and then considering $\hat{\mathbf{x}}_i$ as outlier if $u_{i,\mathcal{E}}$ exceeds a certain cutoff, usually 2.5 or 3.0 [28]. The $median(\mathcal{E})$ and $mad(\mathcal{E})$ is the sample median and median absolute deviation about the median of $\mathcal{E}$, respectively. Pythia provides $\hat{\mathbf{x}}$ to each $S_i \in \mathcal{S}''$ for updating their signatures; see Section 5.1. If $\mathcal{S}' = \emptyset$, Pythia engages all cohorts; if $\mathcal{S}'' = \emptyset$, Pythia engages all cohorts from $\mathcal{S}'$.

4.3 Pythia asymptotic complexity

In COE, given $\mathbf{i}$ Pythia evaluates '$\mathbf{z} \in P_i$', $\forall P_i \in \mathcal{P}$, i.e., it performs one nearest neighbor (1NN) search for each P_i over $\mathcal{C}_i$. We adopt a d-dimensional tree structure [31] for each P_i over the clusters of $\mathcal{C}_i$. Let $\xi = \frac{1}{m}\sum_{i=1}^{m}|\mathcal{C}_i|$ be the average number of clusters in signature P_i. The corresponding time complexity per input $\mathbf{i}$ in COE is $O(md\log(\xi))$. In BC, we also adopt a d-dimensional tree structure over all clusters from all signatures in $\mathcal{P}$. Given $\mathbf{i}$, Pythia performs a 1NN search with $O(d\log(m\xi))$ time since it searches over all clusters from all signatures $\cup_{i=1}^{m}\mathcal{C}_i$. COE and BC require $O(md\xi)$ space. Pythia requires $O(\ell)$ and $O(1)$ communication with cohorts from $\mathcal{S}'$ and the best cohort in COE and BC schemes, respectively.

5. PYTHIA SIGNATURE UPDATE

5.1 COE signature update

Once Pythia has produced $\hat{\mathbf{x}}$ given an input, it updates $\mathcal{P}$. Only $P_i \in \mathcal{P}$, which correspond to cohorts $S_i \in \mathcal{S}''$, need to be updated. The update of P_i is based on the rule $\mathbf{c}_i^* \leftarrow \mathbf{c}_i^* + \eta_i(\mathbf{z} - \mathbf{c}_i^*)$ where $\mathbf{c}_i^* = \arg\min_{\mathbf{c}\in\mathcal{C}_i} \parallel \mathbf{z} - \mathbf{c} \parallel$, i.e., only the dimensions of $\mathbf{c}_i^*$ are modified, which correspond to the n dimensions of the non-MVs of $\mathbf{x}$. This denotes that no new clusters at P_i are formed after the update w.r.t. $\hat{\mathbf{x}}$, since $\mathbf{z} \in P_i$. The exact update can be locally reflected by $S_i \in \mathcal{S}''$ to its signature in order to be secured against a Pythia break-down situation. The magnitude of change in P_i w.r.t. $\hat{\mathbf{x}}$ is $\delta_i = \eta_i \parallel \mathbf{z} - \mathbf{c}_i^* \parallel$.

Let the sum involving the y moments of the reciprocals of binomial coefficients $F_x^{(y)} = \sum_{k=0}^{x} k^y \binom{x}{k}^{-1}$ for non-negative integers x, y. From [29] we obtain that $F_x^{(0)} = \frac{x+1}{2^{x+1}}\sum_{k=1}^{x+1}\frac{2^k}{k}$ and $F_x^{(1)} = \frac{x}{2}F_x^{(0)}$.

THEOREM 4. *The expected magnitude of change in P_i, $E[\delta_i|S_i \in \mathcal{S}'']$, in COE is bounded above by $\delta_i^{\max} = \eta_i\rho_i(F_d^{(0)} - 2)$ and $\delta_i^{\max} \sim (\frac{2}{d-1} - \frac{1}{2^{d-1}})\eta_i\rho_i$ for very large d.*

PROOF. Consider input $\mathbf{i}$ with $1 \le n \le d-1$ non-MVs and $S_i \in \mathcal{S}''$. The probability of choosing a subset of n out of d dimensions corresponding to non-MVs is $\binom{d}{n}^{-1}$. The expected magnitude of change of P_i is $E[\delta_i] = \sum_{n=1}^{d-1}\binom{d}{n}^{-1}\eta_i \parallel \mathbf{z} - \mathbf{c}_i^* \parallel \le \sum_{n=1}^{d-1}\binom{d}{n}^{-1}\eta_i\rho_i^{(n)} \le \sum_{n=1}^{d-1}\binom{d}{n}^{-1}\eta_i\rho_i = (F_d^{(0)} - 2)\eta_i\rho_i$. The asymptotic expansion of $F_d^{(0)} \sim 2 + \frac{2}{d-1} - \frac{1}{2^{d-1}}$ as $d \to \infty$ (proved in [30]). Hence, $\delta_i^{\max} \sim (\frac{2}{d-1} - \frac{1}{2^{d-1}})\eta_i\rho_i$. □

THEOREM 5. *The expected magnitude of change in $\mathcal{P}$, $E[\delta]$, in COE is bounded above by $\delta^{\max} = \eta^{\max}\rho^{\max}(F_m^{(1)} - 1)(F_d^{(0)} - 2)$ and $\delta^{\max} \sim (m-1)(\frac{2}{d-1} - \frac{1}{2^{d-1}})\eta^{\max}\rho^{\max}$ for very large m and d, where $\eta^{\max} = \max\{\eta_i\}_{i=1}^{m}$, $\rho^{\max} = \{\rho_i\}_{i=1}^{m}$.*

PROOF. The probability that a subset $\mathcal{S}''$ of ℓ cohorts is determined by Pythia is $\binom{m}{\ell}^{-1}$. Hence (from Theorem 4),

$$\begin{aligned} E[\delta] &\le \sum_{\ell=1}^{m}\binom{m}{\ell}^{-1}\sum_{i=1}^{\ell}\sum_{n=1}^{d-1}\binom{d}{n}^{-1}\eta_i\rho_i \\ &\le \eta^{\max}\rho^{\max}\sum_{\ell=1}^{m}\ell\binom{m}{\ell}^{-1}(F_d^{(0)} - 2) \\ &= \eta^{\max}\rho^{\max}(F_m^{(1)} - 1)(F_d^{(0)} - 2) \end{aligned}$$

Since $\lim_{m\to\infty}\frac{F_m^{(1)}}{m} \to 1$ (Theorem 11; [29]) and from Theorem 4, we obtain $\delta^{\max} \sim (m-1)(\frac{2}{d-1} - \frac{1}{2^{d-1}})\eta^{\max}\rho^{\max}$. □

5.2 BC signature update

The best cohort S^* updates its signature w.r.t. $\hat{\mathbf{x}}$ as described in Section 5.1, with magnitude of change bounded by $\delta^{*\max}$ (Theorem 4). Note that the change in S^*'s signature is not reflected at Pythia's $\mathcal{P}$ and specifically at the corresponding $P^* \in \mathcal{P}$.

THEOREM 6. *The expected magnitude of change in $\mathcal{P}$, $E[\delta]$, in BC is bounded above by $(F_d^{(0)} - 2)\eta^{\max}\rho^{\max}$.*

PROOF. Each $S_i \in \mathcal{S}$ is equally probable to be selected by Pythia as the best cohort given an input. Hence, from Theorem 5 we obtain $E[\delta] \le \sum_{i=1}^{m}\frac{1}{m}(F_d^{(0)} - 2)\eta_i\rho_i \le (F_d^{(0)} - 2)\eta^{\max}\rho^{\max}$. □

Pythia determines a frequency $\propto (F_d^{(0)} - 1)\eta^{\max}\rho^{\max}$ for a batch update of $\mathcal{P}$ by asking from all (previously engaged as best) cohorts to send their updated signatures changes (referring only to modified clusters), provided that they have not changed from the previous batch update. However, a batch update can be avoided once the best cohort sends the final estimate to Pythia for updating $\mathcal{P}$.

6. PERFORMANCE EVALUATION

6.1 Experiments

Setup. We conducted an extensive series of experiments to assess the performance of Godzilla, ACM and Pythia's schemes COE and BC on two real datasets (D1 and D2) and a synthetic dataset (DS). Real datasets are adopted from the UCI Machine Learning Repository [32]. D1 contains $|\mathcal{X}| = 5 \cdot 10^5$ real valued vectors of $d = 90$ corresponding to audio features. D2 contains $|\mathcal{X}| = 5 \cdot 10^4$ real valued vectors of $d = 384$ corresponding to features extracted from Computed Tomography images. Each vector of DS is a 20-dimensional point with the first fifteen dimensions randomly sampled from a Gaussian mixture of five component Gaussian pdfs with equal mixture weights and mean values of each component randomly selected from the uniform distribution $U(0, 15)$. The other five dimensions are drawn, independently, from the univariate Gaussian distribution $\mathcal{N}(0, 1)$. The first fifteen dimensions are informative dimensions, while the rest dimensions are random noises artificially added to test Pythia's capability of predicting $\mathcal{S}'$. For each dataset, we synthetically produce MVs from each $\mathbf{x}_t$ for $t = 1, \ldots, T$ as follows: each dimension $k = 1, \ldots, d$ from $\mathbf{x}_t$ is randomly and independently marked as missing with MV probability q. In this case, we expect $|\mathcal{X}|\sum_{k=1}^{d-1}\binom{d}{k}q^k(1-q)^{d-k}$ points with MVs; we exclude the cases of missing all dimensions or none. We set $q = 0.3$, which is a relatively high probability of MVs per dimension, thus, being able to test Pythia's robustness in terms of accuracy. On average, a signature P_i contains 0.32% of points of cohort's set $\mathcal{X}_i$ (this amount refers to the number of clusters stored in Pythia) using ART with initial learning rate $\eta = 0.2$, which gradually decreases. Moreover, we set the range percentage $\alpha_k = \alpha = 0.1$ for all dimensions in order to construct ρ. We run all experiments 100 times and took their average values for all performance metrics, with a stream of $T = 1000$ inputs. Pythia's schemes and MVAs

(Section 3.2) were written in Matlab. Table 1 summarizes the parameter values used in our experiments.

Parameter	Notation	Value/Range
d	dimensions	{20,90,384}
α	vigilance range pct.	0.1
η	init. learning rate	0.2
q	MV probability	0.3
m	number of cohorts	$\{5, 10, 20, 50, 100\}$
T	number of inputs	1000

Table 1: Experiment parameters.

Performance metrics. Our metrics include *efficiency metrics* and *accuracy metrics*. A scale-out system consisting of m cohorts affords two types of parallelism: *intra-imputation* and *inter-imputation* parallelism. The former refers to the capability of processing any single imputation using a number of cohorts in parallel, each accessing a dataset partition. The latter refers to the systems' capability of running in parallel a number of imputations, each of which engages a subset of cohorts. It is crucial to note that Godzilla affords neither of these parallelism types and that ACM affords only intra-imputation parallelism. This latter scenario is particularly important as typically a system is presented with a (large) batch of (vector-) inputs, each with missing values and the goal is to impute all input vectors in the batch as quickly/scalably as possible. Given this, our efficiency metrics embody various efficiency aspects impacting scalability. First, we report on *imputation latency*, defined as the time (in seconds) a system (i.e., Godzilla, ACM, Pythia-COE, or Pythia-BC) requires to impute a single input (vector) using a MVA. The rate of latency increase as dataset sizes grow is a strong aspect of scalability. In ACM, latency refers to the time a single cohort requires to impute a single input on its local dataset partition, assuming m cohorts run in parallel. In Pythia, latency refers to the time for COE / BC to predict best cohort(s) $\mathcal{S}''/S^*$, plus the latency to run MVA in parallel at cohort(s). *Imputation speedup* is defined as the ratio of Godzilla latency over ACM / COE / BC latency; it indicates how much a system is faster than Godzilla for a single imputation. *Imputation throughput* is defined as the rate of imputations delivered by a system (number of imputations per second) given a finite stream (batch) of T inputs: with this we capture the inter-imputation parallelism, *in addition to* the intra-imputation parallelism.

We measure *imputation accuracy* using the RMSE metric (root-mean squared difference) between $\mathbf{x}_a$ and $\hat{\mathbf{x}}$:

$$RMSE = \left(\frac{1}{T} \sum_{t=1}^{T} \frac{\sum_{k=1}^{d} w_{tk} (x_{(a)tk} - \hat{x}_{tk})^2}{\sum_{k=1}^{d} w_{tk}} \right)^{1/2}. \quad (4)$$

6.2 Performance results

6.2.1 Imputation efficiency

Fig. 3(a-b) shows the imputation speedup against m for all systems using KNN and REG over D2. Similar results are obtained for D1 which are omitted due to space limitations. Overall ACM, COE and BC achieve an almost linear speedup using both REG and KNN. The speedup of COE and BC drops slightly as m increases since higher m implies more signatures to be processed at Pyhtia.

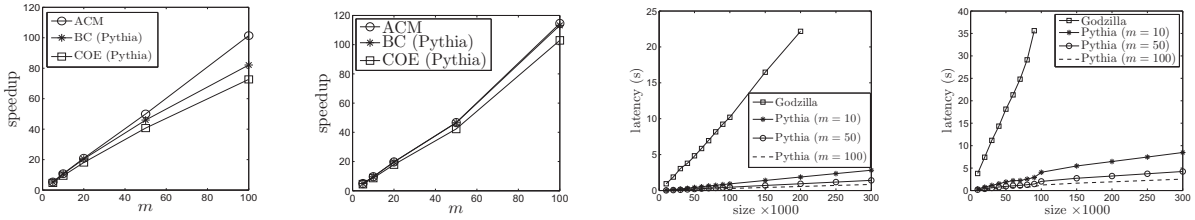
Fig. 3(c-d) shows the latency of Godzilla and Pythia-COE (Pythia-BC curves are very close to Pythia-COE) using REG when $m = \{10, 50, 100\}$ and the size of D1 and D2 varies from 5000 to 300,000 points; (similar results exist for KNN, but are omitted for space reasons). Godzilla struggles with increasing dataset sizes: with over 200,000 and 100,000 points, a high latency over 20s and 35s per input for D1 and D2, respectively, is observed. Pythia scales nicely with its latency per input increasing linearly. Moreover, when the number of cohorts increases, we obtain a sublinear increase in latency. Pythia can easily handle large datasets if more cohorts are available to scale to big data missing values.

Fig. 4(a-b) shows the throughput of each system indicating the capability of handling a stream of T inputs. COE engages S' for an input (or S^* in case of BC) thus the other cohorts ($\in S \setminus S'$) are available to be potentially engaged for other inputs in the stream. Now, recall Fig. 2(b) which shows the average number of cohorts engaged by COE per input for all data sets. For $m = 100$, about 26% of cohorts (average for all data sets) are engaged per input. Obviously, the distribution of the engaged cohorts plays an important role. That is, for a stream of inputs heading for imputation, we achieve very high throughput when (i) $|S'|$ is relatively small (in case of COE) and (ii) different imputations engage different subsets of cohorts. On the other hand, in ACM, all cohorts are concurrently occupied by the same input. The impact of the cohort engagement policy of Pythia's schemes on the throughput is illustrated in Fig. 4(a-b) using REG, where the y-axis is plotted in logarithmic scale for readability. (Similar results exist with KNN). Pythia can handle up to tens of thousands of inputs per second, compared to ACM and Godzilla, which deal with tens of inputs and a few inputs, respectively. As expected, Pythia achieves higher throughput as m increases, as the possibilities for intra-imputation parallelism increase. However, note that in Fig. 4(a) as m increases, we do not achieve further significant increase in throughput, because Pythia's processing over signatures becomes significant. The latter is higher for higher dimensions. In Fig. 4(b), as m increases, Pythia achieves high throughput. We can observe the impact of the number of dimensions d on throughput. D2 contains points with 326% more dimensions than those in D1. Pythia achieves a throughput over 10^4 (inputs/sec) with $m = 20$ in D1, while it achieves the same throughput with $m = 100$ in D2 (five times more cohorts).

Our results up to now clearly make a strong case for the scale-out advantages of the Pythia framework.

6.2.2 Imputation accuracy

Fig. 4(c-d) shows the RMSE against m using KNN and REG on synthetic data. COE and BC, as anticipated based on discussions of Example 1 and 2, obtain significant lower RMSE than Godzilla and ACM. However, this occurs with decreasing benefits as the number of cohorts increases; for $m > 50$ no further decrease in RMSE is achieved. Specifically, COE predicts a subset of cohorts, out of m cohorts, which achieves quite similar RMSE as that obtained by a subset of cohorts out of m' with $m' > m > 50$. In addition, BC engages the best cohort whose estimate is very close to the aggregate estimate of the subset of cohorts engaged by COE. Please note that ACM may yield a higher RMSE depending on the MVA used, even compared to Godzilla. For instance, using KNN, Godzilla would provide the *global* best



(a) Speedup; (KNN) (b) Speedup; (REG) (c) Latency; (REG) (d) Latency; (REG)

Figure 3: (a-b) Imputation speedup against m of ACM, COE and BC in dataset D2 using KNN and REG; (c-d) Imputation scalability against m of Godzilla and Pythia (COE) in datasets D1 & D2 using REG.

K points, whereas in ACM each cohort, even when storing irrelevant data, will be contributing its best K points. The latter necessarily implies that ACM's imputation involves points which adversely affect imputation errors.

Fig. 5 shows the RMSE against m using KNN and REG on real datasets. Pythia's schemes achieve comparable RMSE with Godzilla, with COE assuming relatively the lowest RMSE for both MVAs and datasets. In addition, the RMSE of COE remains at its lowest value from a certain m value (e.g., $m = 50$ in D2) thus there is no need to involve more cohorts. BC performs slightly better than Godzilla for both MVAs and datasets. Moreover, BC assumes higher RMSE than COE. This denotes the robustness of COE compared to BC in terms of accuracy due to the aggregate estimate from multiple engaged cohorts. ACM has higher RMSE than Godzilla in both datasets since it aggregates the estimates of all cohorts possibly incorporating estimates that spoil the final result.

6.3 Discussion

The central conclusions of our study are the following:

- Godzilla suffers from obvious severe scalability / efficiency limitations. Furthermore, it can have a poor performance even in terms of imputation accuracy.
- ACM offers efficiency performance comparable to what Map-Reduce solutions to scalability would offer, in that it requires all cohorts to be engaged for MV imputation. As such, it can only improve per-imputation efficiency. Our results show that ACM performs poorly in terms of both MV imputation throughput (compared to Pythia) and accuracy (compared to Pythia *and even* Godzilla).
- Pythia is a great all-around performer, significantly outperforming both ACM and Godzilla in terms of both overall efficiency and accuracy. Note that, even though ACM enjoys a smaller per-imputation latency than Pythia, this is achieved at a significant cost for overall imputation throughput and accuracy.
- Finally, the two Pythia schemes BC and COE, as expected can trade-off efficiency for accuracy with BC offering higher throughput but at lower accuracy.

7. CONCLUSIONS

We have tackled the problem of scaling out MV imputations, a common problem in many big data applications. We studied and developed some of the fundamentals of the problem, based on which we developed Pythia, a framework and algorithms designed for this aim. The Pythia framework is drastically different, as it on the one hand avoids the need to access all cohorts (and all associated costs for communication and for running MVAs at all cohorts), while on the other can achieve better or comparable MV imputation accuracy, compared to centralized solutions. Specifically, our comprehensive experiments showed that it can provide drastically better efficiency/scalability and accuracy compared to a centralized approach (Godzilla) and a massively parallel, a la Map-Reduce, solution (ACM). Future work plans entail the study of additional cohort prediction schemes, straddling the line between efficiency and accuracy.

8. ACKNOWLEDGEMENTS

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program 'Education and Lifelong Learning' of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

9. REFERENCES

[1] X. Su, *et al*, 'Using Classifier-Based Nominal Imputation to Improve Machine Learning', *Proc. 15th PAKDD*, Part I, LNAI 6634, pp. 124–135, 2011.

[2] A. Farhangfar, *et al*, 'Impact of imputation of missing values on classification error for discrete data', *Pattern Recognition*, 41(12): 3692–3705, Dec 2008.

[3] M.T. Asif, *et al*, 'Low–Dimensional Models for Missing Data Imputation in Road Networks', *Proc. 38th IEEE ICASSP*, pp.3527–3531, 2013.

[4] E.C. Chi, *et al*, 'Genotype imputation via matrix completion', *Genome Research*, 23(3):509–18, Mar 2013.

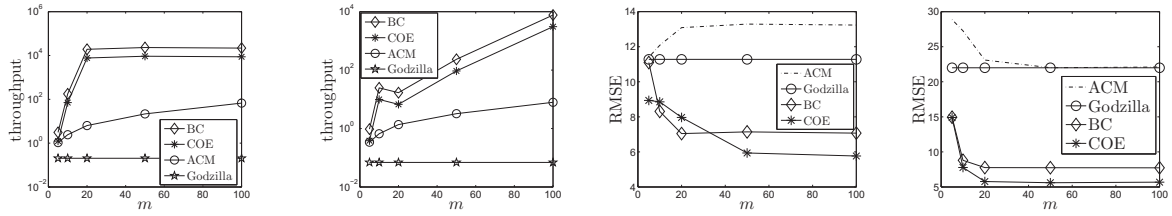
[5] I.B. Aydilek, *et al*, 'A novel hybrid appoach to estimating missing values in databases using k–nearest neighbors and neural networks', *Innovative Computing, Information and Control*, 8(7A): 1349–4198, Jul 2012.

[6] A. Farhangfar, *et al*, 'A Novel Framework for Imputation of Missing Values in Databases', *IEEE Trans. Sys. Man Cyber. (A)*, 37(5): 692–709, Sep 2007.

[7] K. Lakshminarayan, *et al*, 'Imputation of missing data in industrial databases', *Appl. Intell.*, 11(3): 259–275, Nov / Dec 1999.

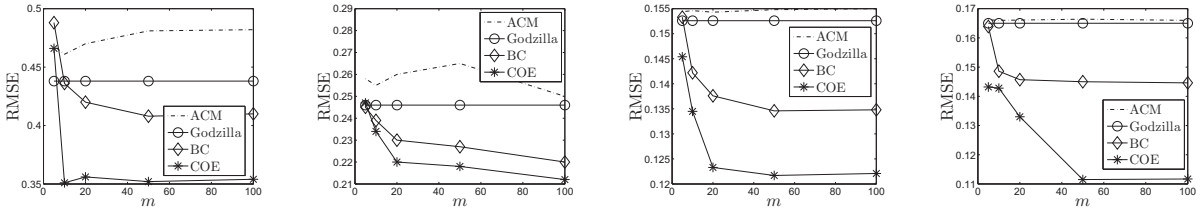
[8] L. A. Kurgan, *et al*, 'Mining the cystic fibrosis data', J. Zurada & M. Kantardzic (Eds.), *Next Generation of Data–Mining Applications*, IEEE Press, 415–444, 2005.

[9] A.W. Liew, *et al*, 'Missing value imputation for gene expression data: computational techniques to recover missing data from available information', *Brief. Bioinform.*, 12(5): 498–513, Sep 2011.



(a) Throughput; REG (b) Throughput; REG (c) RMSE; KNN (d) RMSE; REG

Figure 4: (a-b) System throughput against m of Godzilla, ACM, COE and BC in dataset D1 & D2 using REG; (c-d) RMSE against m of Godzilla, ACM, COE and BC in dataset DS using KNN and REG.



(a) RMSE; D1 (KNN) (b) RMSE; D1 (REG) (c) RMSE; D2 (KNN) (d) RMSE; D2 (REG)

Figure 5: RMSE against m of Godzilla, ACM, COE and BC in dataset D1 (a-b) and D2 (c-d) using KNN and REG.

[10] J. Dean, *et al*, 'MapReduce: Simplified Data Processing on Large Clusters', *Proc. USENIX OSDI*, 2004.

[11] S. Ghemawat, *et al*, 'The Google File System', *Proc. ACM SOSP*, 2003.

[12] C-T. Chu, *et al*, 'Map-Reduce for Machine Learning on Multicore', *NIPS 19*, MIT press, 281–288, 2006.

[13] C. K. Enders, 'Applied Missing Data Analysis', *Guilford Press*, NY, 2010.

[14] D. W. Joenssen, *et al*, 'Hot Deck Methods for Imputing Missing Data', *Proc. 8th MLDM* , LNCS 7376, pp.63–75, 2012.

[15] O. Troyanskaya, *et al*, 'Missing value estimation methods for DNA microarrays', *Bioinformatics*, 17(6):520–525, 2001.

[16] R.J. Little, *et al*, 'Statistical Analysis with Missing Data', *Wiley*, NY, 1987.

[17] T.E. Raghunathan, *et al*, 'A multivariate technique for multiply imputing missing values using a sequence of regression models', *Survey Methodology*, 27(1):85–95, 2001.

[18] D.B. Rubin, 'Multiple Imputation After 18+ Years', *J. of the American Statistical Association*, 91(434):473–489, 1996.

[19] L. Li, *et al*, 'DynaMMo: mining and summarization of coevolving sequences with missing values', *Proc. 15th KDD*, 527–534, 2009.

[20] S. Yang, *et al*, 'Online recovery of missing values in vital signs data streams using low–rank matrix completion', *Proc. 11th IEEE ICMLA*, 281–287, 2012.

[21] M. Ouyang, *et al*, 'Gaussian mixture clustering and imputation of microarray data', *Bioinformatics*, 20(6): 917–923, Apr 2004.

[22] T. Aittokallio, *et al*, 'Dealing with missing values in large-scale studies: microarray data imputation and beyond' *Brief. Bioinform.* 11(2):253–264, 2010.

[23] D-W. Kim, *et al*, 'Iterative Clustering Analysis for Grouping Missing Data in Gene Expression Profiles', *Proc. PAKDD 2006*, LNAI 3918, pp.129–138, 2006.

[24] M.R. Garey, *et al*, 'Computers and Intractability; A Guide to the Theory of NP–Completeness', *W. H. Freeman & Co.*, NY, 1990.

[25] B. Przydatek, 'A fast approximation algorithm for the subset–sum problem', *Intl. Trans. in Op. Res.*, 9(4): 437–459, Jul 2002.

[26] G. A. Carpenter, *et al*, 'The ART of adaptive pattern recognition by a self–organizing neural network', *IEEE Computer*, 21(3): 77–88, Mar 1988.

[27] A. Ahmad, *et al*, 'A k–mean clustering algorithm for mixed numeric and categorical data' *Data & Knowledge Engineering*, 63(2):503–527, 2007.

[28] P. J. Rousseeuw, *et al*, 'Alternatives to the median absolute deviation', *J. American Statistical Association*, 88(424): 1273–1283, Dec 1993.

[29] H. Belbachir, *et al*, 'Sums involving moments of reciprocals of binomial coefficients', *J. Integer Sequences*, 14(6), Article 11.6.6, 16p, 2011.

[30] J-H. Yang, *et al*, 'The asymptotic expansions of certain sums involving inverse of binomial coefficient', *Intl. Mathematical Forum*, 5(16): 761–768, 2010.

[31] J.L. Bentley, 'Multidimensional binary search trees used for associative searching', *Communications of the ACM*, 18(9):509–517, 1975.

[32] K. Bache, *et al*, UCI Machine Learning Repository [http://archive.ics.uci.edu/ml] Irvine, Uni. of California, School of Inform. and Comp. Sci., 2013.

Rank Join Queries in NoSQL Databases

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

Ioannis Patlakas
Max-Planck-Institut für Informatik, Germany
patlakas@mpi-inf.mpg.de

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

ABSTRACT

Rank (i.e., top-k) join queries play a key role in modern analytics tasks. However, despite their importance and unlike centralized settings, they have been completely overlooked in cloud NoSQL settings. We attempt to fill this gap: We contribute a suite of solutions and study their performance comprehensively. Baseline solutions are offered using SQL-like languages (like Hive and Pig), based on MapReduce jobs. We first provide solutions that are based on specialized indices, which may themselves be accessed using either MapReduce or coordinator-based strategies. The first index-based solution is based on inverted indices, which are accessed with MapReduce jobs. The second index-based solution adapts a popular centralized rank-join algorithm. We further contribute a novel statistical structure comprising histograms and Bloom filters, which forms the basis for the third index-based solution. We provide (i) MapReduce algorithms showing how to build these indices and statistical structures, (ii) algorithms to allow for online updates to these indices, and (iii) query processing algorithms utilizing them. We implemented all algorithms in Hadoop (HDFS) and HBase and tested them on TPC-H datasets of various scales, utilizing different queries on tables of various sizes and different score-attribute distributions. We ported our implementations to Amazon EC2 and "in-house" lab clusters of various scales. We provide performance results for three metrics: query execution time, network bandwidth consumption, and dollar-cost for query execution.

1. INTRODUCTION

Cloud stores have become the storage of choice for a large variety of big data producers, consumers, and managers (e.g., Twitter, Facebook, Google, Amazon, etc.) For many modern Big Data applications, RDBMSs were found lacking, particularly with respect to scalability (in terms of number of data items, users, operations per second, etc.), despite valiant efforts (e.g., sharding, memory caches, partial denormalization, etc.). To fill this gap, two complementary technologies emerged: NoSQL databases and the MapReduce framework. Interestingly, even traditional key RDBMS players, such as Oracle, are now focusing on NoSQL products coupled with MapReduce platforms. There is an impressive list of NoSQL cloud databases; e.g., BigTable, DynamoDB, Riak, HBase, Cassandra, etc. All these are purpose-built to scale across a large number of servers (by sharding/horizontal partitioning of data items), to be fault tolerant (through replication, write-ahead logging, and data repair mechanisms), to achieve high write throughput (by employing memory caches and append-only storage semantics) and low read latencies (through caching and smart storage data models), flexibility (with schema-less design and denormalization), and development friendliness (Object Relational Mappings are typically avoided, etc.). Further, different systems offer different approaches to issues such as consistency, replication strategies, data types, and models.

The data model employed by NoSQL DBs can be viewed as a key-value model, built around four core abstractions: (a) the "key-value pair": a quadruplet {key, column name, column value, (and perhaps a timestamp)}, uniquely identified by the combination of key, column name (and timestamp, where applicable); (b) the "table": an ordered collection of key-value pairs; and (c) the "row": a collection of all key-value pairs in a given table, sharing the same key. Some systems further employ the notion of "column families", in essence partitioning the table data vertically so that each such family includes key-value pairs with specific column names. All such systems support efficient point queries and sequential scans on key-value pairs and rows based on their key, as well as efficient insertion/deletion of key-value pairs. However, queries on other parts of the data (e.g., column names/values, timestamps, etc.) are costly, often requiring a scan of all key-value pairs even for simple equality queries.

Despite original intents, NoSQL is now "re-baptized" to spell "Not Only SQL". As more data poured into these systems, the need for complex queries – particularly for analytics – emerged, leading to the rise of data warehousing systems, such as Hive[24] and Pig[21], offering SQL-like interfaces. This went hand-in-hand with the realization of the shortcomings of denormalization for several real-world analytic workloads; carrying denormalization to the extreme implies a huge amount of data being repeated across very large numbers of rows in a "universal" table, creating an update/maintenance nightmare and utterly negating several of the key performance advantages of NoSQL systems in the long run. Although NoSQL guarantees fast keyed-row retrievals, these rows now contain typically lots of data use-

Proceedings of the VLDB Endowment, Vol. 7, No. 7

less to most queries, resulting in poor query performance as many more disk IOs are needed to get the desired data (e.g., in typical queries involving large scans of rows), and more useless data are shipped across the NoSQL cloudstore substrate. Hence, inevitably NoSQL systems transformed to contain several tables, linked through foreign-key-like constructs (although without the consistency semantics inherent in classic RDBMSs) and joined at query time. The issue is that the burden to do this lies on the applications. This is currently the conventional wisdom, especially when modeling many-to-many relationships and storing them in NoSQL systems. Orthogonally, emerging analytic tasks often rely on data dispersed across different files or tables.

Motivations. As joins are very resource hungry, several recent research efforts have attempted to expedite them in cloud stores. So far, ranked (top-k) equi-joins have been completely overlooked in this setting, despite the fact that these queries arise naturally in a variety of situations, as joining and ranking are fundamental to many analytics tasks. Take for example a collection of per-day search engine logs, consisting of phrases and their frequency of appearance in user inputs, with a separate table or file per day. Now imagine we wish to find the k most popular phrases appearing in several of these days. This would be formulated as a rank-join query, where the phrase text is the join attribute, and the total popularity of each phrase is computed as an aggregate over the per-day frequencies. Rank equi-joins also arise in full-text search scenarios. Imagine a collection of posting lists over a large text corpus consisting of several documents. Each posting list would include information for documents containing the related keyword, with each list entry consisting of (at least) the document identifier and the document's relevance score with regard to the keyword. With the size of many of these lists being in the gigabytes even for relatively small collections such as Wikipedia dumps (and much larger when scaling to web archives and/or the actual World-Wide-Web), and given the need to scan through them efficiently, it is only reasonable to assume that each list is stored in a separate table in a key-value store. Then, finding the most relevant documents for two (or more) keywords, consists of a rank-join over the corresponding posting lists, where the document ID is the join attribute and the relevance of each document to the search phrase is computed using a function over the individual relevance scores.

1.1 Problem Formulation

We assume operation in a distributed shared-nothing cloud store. Please note that the raw table data can be stored either at the filesystem level (e.g., HDFS) or at any NoSQL store. Our algorithms are impervious to this!

Formally, a top-k join query can be written as:

`SELECT` *select-list* `FROM` $R_1, R_2, \ldots, R_n$
`WHERE` *equi-join-expression*$(R_1, R_2, \ldots, R_n)$
`ORDER BY` $f(R_1, R_2, \ldots, R_n)$ `STOP AFTER` k

Scoring of individual rows is typically based on either a (predefined) function on attribute values, or some explicit "score" attribute (e.g., movie rating, PageRank score, etc.). The solutions we discuss make no distinction between these two cases and work equally well for both; however, for ease of presentation and without loss of generality, we assume there is a scoring attribute in each row. Furthermore, for ease of presentation, we assume that score attributes take values in $[0, 1]$; it should be obvious that our algorithms perform the same with arbitrary score values as well, provided that there is a total ordering on these values. Last, as is common in rank-join works, the score of tuples in the join result set is computed using a monotonic aggregate function $f(\cdot)$ on their individual scores.

A naive approach would first compute the join result, then rank and select the top-k tuples; this is the approach of Hive and (with minor optimizations) of Pig. This is obviously extremely costly, even in centralized settings, let alone when data have to be shipped across the network. Our challenge is to compute the result set without producing the full join result, ensuring that the amount of data that is transferred is as little as possible and the query latency is small.

1.2 Contributions

A study of how to efficiently process top-k join queries in NoSQL cloudstores is very much lacking, given the rapid popularity increase of such environments and their unique characteristics. We will use as a reference point the baseline techniques of using Hive or Pig to formulate and execute such queries in a massively parallel manner. However, acknowledging their disadvantages we will contribute and study the performance of a number of different approaches. First, we contribute a MapReduce solution that is based on specialized indices. Second, in the no-MapReduce realm, we contribute an algorithm coined Inverse Score List Rank Join (ISL), which is based on the popular HRJN[13] centralized rank join algorithm. Our third contribution is an algorithm based on a novel rank-join statistical access structure, coined the Bloom Filter Histogram Matrix (BFHM). We prove that the BFHM algorithm can achieve 100% recall (despite its probabilistic nature). We have chosen to store all our access structures in a NoSQL DB. Although not a hard requirement, this choice was dictated by the need to handle possible large volumes of new item insertions/deletions, for which NoSQL DBs are much better suited than DFSs. Further, we also provide algorithms for efficiently maintaining indexes in the face of updates. Our solutions offer trade-offs with regard to various metrics. We thus further contribute an in-depth performance evaluation in real-world systems (Amazon's EC2 and in-house clusters, using TPC-H datasets), against the baseline approaches. The metrics are: query processing time, network bandwidth consumption, and query processing dollar cost (e.g., when executed on charge-per- item infrastructures, such as DynamoDB).

2. RELATED WORK

Rank Queries. Well-known ranking operators include the top-k and kNN operators. Top-k (selection) operators typically accept as input a set of records each of the type $< ID, score_1, \ldots, score_n >$, a monotone score aggregation function, $f(score_1, \ldots, score_n) \rightarrow [0, 1]$ and a threshold number k, and produce the IDs of records whose aggregate score (from all n score attributes) is in the top-k among the score of all records. kNN operators are different; they accept as input a specific record, r, a table of records $T = \{t_1, \ldots, t_n\}$ and a notion of similarity among records $sim(s, t)$, and produce the result set $R = \{t_{R_1}, t_{R_2}, ..., t_{R_k}\}$ with $t_{R_i} \in T$ and $|R| = k$, so that $sim(r, t_{R_i}) > sim(r, t_j)$, $\forall t_j \in T \bigwedge t_j \notin R$.

Our paper focuses on ranking a la top-k queries defined above. A survey of top-k query processing algorithms for

this setting is presented in [14]. In [10] instance-optimal algorithms are presented, i.e., the Threshold Algorithm (TA) and variants, using sorted and/or random accesses. The first notable distributed variant of TA is the Three-Phase Uniform Threshold (TPUT) algorithm[5]. KLEE[16] improved TPUT by employing histograms and Bloom filters for each node to limit the tuple search space and bandwidth consumption. Last, the Threshold Join Algorithm (TJA) [28] is a top-k selection query processing algorithm, using an outer join step to maintain partial top-k results as these are aggregated at parent nodes. TJA is developed for a hierarchical (sensor) topology – a setting drastically different than that of NoSQL clusters.

Rank and Join Queries. *kNN joins*[26], as well as *similarity joins*[4] include both rank and join operators. Similarity joins accept as input two collections of sets, R and S, a notion of set similarity and a threshold similarity value t, and return all pairs (r, s) with $\{r \in R\}$ and $\{s \in S\}$, such that $sim(r, s) > t$. This is equivalent to returning, for each record r from R, all records s from S, such that $sim(r, s) > t$. kNN join operators are similar to the above, generalizing the kNN operator to perform kNN operations not for just a given record but for a set of records. Finally, *top-k similarity joins*[27], like similarity and kNN joins, use a notion of distance to define similarity among records from R with those from S. A top-k similarity join returns those joined record pairs having the highest k similarities. The top-k join operators considered in this paper extend top-k selection queries by aggregating score attributes from two (or more) relations and by returning the top-k scores only of records which are in the result of a join operation (based on some other common attribute of the two relations). As such, top-k join queries are substantially different than kNN join and (top-k) similarity join queries. Joining is performed on separate explicit join attributes (introducing challenges/opportunities for extra indexing and optimisations) and ranking is based on a monotone aggregate function of the relations' score attributes (as opposed to a distance between records).

Ilyas et al. presented NRA-RJ [12] to support such rank joins. A pipelined operator, J^*, was presented in [19], providing a join operator with a general non-equi-join condition. In [13], Ilyas et al. present an influential algorithm for ranked join (HRJN) (to be discussed at more length later). More recently, [22] presented the Pull/Bound Rank Join (PBRJ) algorithm, a generalization of HRJN-style algorithms. [25] proposes a join graph in which joining attributes are represented by nodes and relations among attributes by edges. This approach provides support for top-k join queries (other than inner-join ones) over web databases.

Join Queries on Clouds. Recently join queries in the cloud received considerable attention. Hive [24] and Pig [21] support joins over very large datasets using MapReduce. HadoopDB[1] replaces local data stores with DBMS instances and supports the execution of MapReduce jobs over them; in essence, it is a parallel DBMS enriched with MapReduce capabilities, and is unlike NoSQL systems. Hadoop++[7] proposes "Trojan indices", created at data load time on join attributes and collocated with the data read by each mapper, allowing mappers to avoid expensive disk scans to sort data. Then, "Trojan Joins" co-partition the splits of the relations to be joined, yielding map-only jobs and saving considerable overhead. Results have also been produced for other than equi-joins and 2-way joins [2, 15, 20].

Bloom Filters, Joins, and Top-k. A Bloom filter (BF)[3, 17] is a data structure that compactly represents a set of items as a bit vector to expedite membership queries. [6, 18] use BFs to estimate the cardinality of a join result. Our treatment of BFs differs in its statistical intuition from both [18] and [6]. Finally, KLEE also uses BFs and histograms for top-k queries; however, KLEE does not deal with joins, the actual data structures and their use are different, it doesn't use counting filters, and cannot guarantee 100% recall.

Rank Joins in Distributed Settings. Although inspirational, none of the above works have attempted to solve the problem of top-k equi-join queries in cloud stores. [29] and [8] tackled the problem of rank join processing in large distributed systems. They both attempt to compute a bound on the scores of individual tuples from the base relation, in order to prune tuples not participating in the top-k join result, and both assume operation over a DHT network overlay. [29] uses a sampling stage in which the querying node multicasts the query to a random set of peers. These peers then perform a hash-join by rehashing their data onto the DHT using the join value as the hash function input. The querying node collects the result set and retains the k'th highest score. It then broadcasts this score to all nodes, which in turn perform a distributed hash-join again, only now limiting the rehashed items to those that can produce a join result with a score above the threshold (assuming they join with a tuple with the maximum score value). [8] employ 2-D equi-width histograms; for each of the distinct join values in the input data, they build and store a histogram on the corresponding score values. Query processing consists of two stages – score bound estimation using the histogram buckets, and pulling of data tuples with scores above the bound – repeated in sequence until the final result is produced. As maintaining one bucket per distinct join value is not feasuble in real scenarios, the authors generalize their solution by grouping same-score buckets for adjacent join values and combining them using the uniform frequency assumption. Both of these approaches (much like our ISL and BFHM), fall in the general family of PBRJ-style algorithms, interchanging between bound computation and data pulling. Both these and ISL produce bounds on the tuple scores, ignoring however their join attribute values, thus ending up transferring more tuples than necessary (as several of them may not contribute to the final result due to not joining with any other tuple). Such approaches are at a disadvantage in cloudstores, as their processing time is dominated by data transfers. This situation is further aggravated by the fact that sampling ([29]) and approximate statistics ([8]) often lead to inaccurate estimations and either low recalls (e.g., as low as 0.2 for [8]) or extremely high query processing costs (to be shown shortly). BFHM, however, combines histograms with Bloom filters to locate tuples that will indeed end up in the final result set, achieving further savings in query processing time, bandwidth consumption, and dollar-cost, while guaranteeing a perfect 100% recall.

3. BASELINE RANK JOINS

We focus on two-way equi-joins; extending the algorithms

R_1			R_2		
row key	join value	score value	row key	join value	score value
r_{1_1}	d	0.82	r_{2_1}	a	0.51
r_{1_2}	c	0.93	r_{2_2}	b	0.91
r_{1_3}	c	0.67	r_{2_3}	c	0.64
r_{1_4}	d	0.82	r_{2_4}	d	0.53
r_{1_5}	a	0.73	r_{2_5}	d	0.41
r_{1_6}	c	0.79	r_{2_6}	d	0.50
r_{1_7}	b	0.82	r_{2_7}	a	0.35
r_{1_8}	b	0.70	r_{2_8}	a	0.38
r_{1_9}	d	0.68	r_{2_9}	a	0.37
$r_{1_{10}}$	a	1.00	$r_{2_{10}}$	c	0.31
$r_{1_{11}}$	b	0.64	$r_{2_{11}}$	b	0.92

Figure 1: Running example: Tuples of R_1 and R_2

to multi-way joins is straightforward. We first describe Hive's and Pig's approaches, as the baseline solutions for rank-join queries. Fig. 1 shows the input for our running example.

3.1 Rank Joins with Hive and Pig

In Hive, rank join processing consists of two MapReduce jobs plus a final stage. The first job computes and materializes the join result set, while the second one computes the score of the join result set tuples and stores them sorted on their score; a third, non-MapReduce stage then fetches the k highest-ranked results from the final list.

Pig takes a smarter approach. Its query plan optimizer pushes projections and top-k (STOP AFTER) operators as early in the physical plan as possible, and takes extra measures to better balance the load caused by the join result ordering (ORDER BY) operator. Specifically, 3 MapReduce jobs are used. The first computes the join result: mappers scan the table files, do early projections (stripping out unrelated columns), and emit rows with the join value as their key; then, reducers group together rows with the same join value, produce the join result set, and store it in an HDFS file. The second MapReduce job is a by-product of the ORDER BY clause; it samples the records in the join result file in the map phase, and appropriate quantiles are computed at the reduce phase. These quantiles are then used to construct a balanced partitioner for the third job, which orders the temporary records on their score and produces the top-k result set. First, the map phase emits the temporary records with their join score as their key, and a combiners take over producing a local top-k list. These lists are then assigned to a sole reducer producing the final top-k result set.

4. INDEXED RANK JOINS

As BigTable/HBase were the archetypical key-value cloud-stores, we borrow their terminology in the description of the various algorithms and examples. It should be clear, though, that our indices and algorithms apply (perhaps with slight, obvious modifications) to all contemporary key-value stores.

4.1 Inverse Join List MapReduce Rank-Join

Our first algorithm – Inverse Join List MapReduce rank join (IJLMR) – uses MapReduce, but utilizes an index to reduce the required MapReduce jobs to one, and avoid extra network transfers and inefficiencies.

4.1.1 IJLMR Index

In the above approaches, the first stage mappers actually create an inverted list of input tuples keyed by their join values; this is, in essence, a materialized view where each

Algorithm 1 IJLMR Index Creation

```
1 Input: Rows from a single column family (e.g., A), of
      the form {row.rowKey: row.joinValue, row.score}
2 Output: IJLMR index rows for A
3 Map():
4   foreach(row ∈ sequential scan of A)
5     emit(row.joinValue: row.rowKey, row.score);
```

Row key (join value)	Index tuples ({row key, score})	
	R_1	R_2
a	{$r_{1_{10}}$,1.00}, {r_{1_5},0.73}	{r_{2_1},0.51}, {r_{2_7},0.35}, {r_{2_8},0.38}, {r_{2_9},0.37}
b	{r_{1_7},0.82}, {r_{1_8},0.70}, {$r_{1_{11}}$,0.64}	{r_{2_2},0.91}, {$r_{2_{11}}$,0.92}
c	{r_{1_2},0.93}, {r_{1_3},0.67}, {r_{1_6},0.79}	{r_{2_3},0.64}, {$r_{2_{10}}$,0.31}
d	{r_{1_1},0.82}, {r_{1_4},0.82}, {r_{1_9},0.68}	{r_{2_4},0.53}, {r_{2_5},0.41}, {r_{2_6},0.50}

Figure 2: Running example: IJLMR index table

entry has a join value as its key, and the input rows with that specific join value as its set of values. Our IJLMR index consists of a space-optimized form of these inverted lists, where index values consist of a list of tuples each being a combination of the row key and score value of the indexed row (Fig. 2). The IJLMR index for each indexed table is stored as a separate column family in one big table. This means that, if the table is split up/sharded and distributed across the NoSQL store nodes, index entries for the same join values across all indexed tables are stored next to each other on the same node. The IJLMR index is built with a map-only MapReduce job (Algorithm 1) – a special type of MapReduce job where there are no reducers and the output of mappers is written directly into the NoSQL store. We also provide routines for online maintenance and updates to this index, to be discussed shortly.

4.1.2 IJLMR Query Processing

The IJLMR query processing algorithm consists of a single MapReduce job/stage, with several mappers and a single reducer (Algorithm 2). In this job, each mapper scans through its partition of the IJLMR index, reading columns from the index column families for the joined tables, one row at a time. For each row, it computes the Cartesian product (i.e., the join result) and join score of index entries from the different column families; e.g., the mapper responsible for join value a (see Fig. 2) would produce 2×4 key-values, the mapper responsible for b would produce 3×2 tuples, etc. The mappers store in-memory only the top-k ranking result tuples, and emit their final top-k list when their input data is exhausted. The single reducer then combines the individual top-k lists and emits the global top-k result.

In addition to reducing the overall MapReduce jobs and stages down to one, this design has the added benefit that data transfers due to MapReduce shuffling/sorting are minimized; the Hadoop framework ensures that each mapper is executed on the NoSQL store node storing its input region data (or as close to it as possible), and thus only the individual top-k result sets are transferred across the network and shuffled/sorted at the single reducer. As we shall see in the performance evaluation section, this approach achieves at least an order of magnitude faster query processing times compared to Hive, and several orders of magnitude

Algorithm 2 IJLMR Rank-Join

```
1  Input: Rows from the IJLMR index for column families A
       and B, of the form {row.joinValue: row.rowKey,
       row.score}
2  Output: Top-k join result set
3  Map():
4    foreach(row ∈ input) {
5      HashTable tuplesA=∅, tuplesB=∅;
6      SortedList results=∅;
7      foreach(kv ∈ row) {
8        if (kv.columnFamily == A) {
9          myTuples = tuplesA.get(kv.joinValue);
10         otherTuples = tuplesB.get(kv.joinValue);
11       } else {
12         myTuples = tuplesB.get(kv.joinValue);
13         otherTuples = tuplesA.get(kv.joinValue);
14       }
15       foreach (kv′ ∈ otherTuples) {
16         results.add(innerJoin(kv, kv′));
17         results.trim(k);
18       }
19       myTuples.append(kv.joinValue, kv);
20     }
21     emit(results);
22 Reduce():
23   SortedList results=∅;
24   foreach(kv ∈ input) {
25     results.add(kv);
26     results.trim(k);
27   }
28   emit(results); // Final top-k result set
```

less bandwidth consumption. Unfortunately, note that the mappers still have to scan through the entire input dataset, weighing on the dollar-cost of query processing, which is almost as high as that of the Hive approach.

4.2 Inverse Score List Rank-Join

Clearly, network and disk I/O bandwidth savings are tantamount. Also, taking advantage of the intrinsics of the query processing engine of the NoSQL store at hand can provide further improvements. We note that MapReduce is a poor match for such complex queries as top-k joins. Our intuition is similar to that of Stonebraker, et al.[23], who pointed out that MapReduce is suboptimal for several aspects of data management, including complex queries. Following this thread of thought, we first overview HRJN[13] and then contribute Inverse Score List rank join (ISL).

4.2.1 HRJN Overview

Assume an n-way rank join between relations R_1, R_2, ..., R_n. In HRJN the tuples of each relation R_i are sorted in lists, ranked according to a scoring attribute $R_i.score$, or by using a scoring function on one or more attribute values. For simplicity, we assume the former scenario, but our solutions are equally applicable in the latter. The score of the n-way join result tuples is then computed using a monotonic ranking function $f(R_1.score, \ldots, R_n.score)$ on the individual scores of joined tuples. Tuples from the n lists are iteratively retrieved in decreasing score order, and the algorithm keeps the minimum ($\bar{s}_i$) and maximum ($\hat{s}_i$) tuple scores seen thus far (for $i \in [1, \ldots, n]$). Every retrieved tuple is joined against previously retrieved ones and appended to the result set, if the latter has less than k tuples or the score of the new join tuple is higher than that of the k^{th} tuple (resulting in the latter's elimination). Then, a threshold score S is computed as: $S = max\{f(\bar{s}_1, \hat{s}_2, \ldots, \hat{s}_n),\ f(\hat{s}_1, \bar{s}_2, \ldots, \hat{s}_n),\ \ldots f(\hat{s}_1, \hat{s}_2, \ldots, \bar{s}_n)\}$. In other words, the threshold score equals the maximum attainable score by any subsequent join result tuple. The algorithm terminates when the score of the k^{th} join result is greater than the threshold.

Algorithm 3 ISL Index Creation

```
1  Input: Rows from a single column family (e.g., A), of
       the form {row.rowKey: row.joinValue, row.score}
2  Output: ISL index rows for A
3  Map():
4    foreach(row ∈ sequential scan of A)
5      emit(row.score: row.rowKey, row.joinvalue);
```

Row key (score)	Index tuples ({row key, join value})	
	R_1	R_2
-1.00	$\{r1_{10}, a\}$	
-0.93	$\{r1_2, c\}$	
-0.92		$\{r2_{11}, b\}$
-0.91		$\{r2_2, b\}$
-0.82	$\{r1_1, d\}$, $\{r1_4, d\}$, $\{r1_7, b\}$	
-0.79	$\{r1_6, c\}$	
	...	
-0.35		$\{r2_7, a\}$
-0.31		$\{r2_{10}, c\}$

Figure 3: Running example: ISL index table

4.2.2 The ISL Index

Like HRJN, ISL is based on the existence of inverted score lists. These lists are part of the ISL index, created via a map-only MapReduce job (Algorithm 3), just like in the case of the IJLMR index above. More specifically, for each input relation, we build and maintain an index, comprised of a column family in a common index table, where each row has a score value as its key, and the set of input tuples with this value in their score attribute as the content of the row. A kink of HBase is that it provides fast scans in increasing rowkey order but has no support for scans in the other direction; due to this, in our implementation we have used the negated score values as the index keys (see Fig. 3).

4.2.3 ISL Query Processing

The query processing algorithm is outlined in Algorithm 4. During query processing, the "coordinator" scans through the index column families for the joined relations alternately. Scanning is performed in increasing key (i.e., decreasing score) order, and in batches of a user-defined size. As NoSQL stores are in essence column stores, and key-value pairs with subsequent keys are stored next to each-other on disk, batching reads results in a lower disk I/O overhead, as well as a lower processing time due to the cost of IPC calls to the NoSQL store being amortized over the batch size. As we shall see in the performance evaluation section, such batched scans (e.g., HBase scans with a non-zero rowcache size) can result in significant gains in query processing times, trading off bandwidth consumption and dollar-costs. The coordinator stores (in-memory) all retrieved tuples in separate hash tables, using the join value as the key; this allows for fast joins whenever new tuples are fetched. The coordinator further maintains a list of the current top-k results. With every new tuple fetched and processed, the coordinator computes

Algorithm 4 ISL Rank-Join

```
1  Input: Rows from the ISL index for column families A
       and B, of the form {row.score: row.rowKey,
       row.joinvalue} (see Fig. 3), batch sizes C_A, C_B
2  Output: Top-k join result set
3  HashTable tuplesA=∅, tuplesB=∅;
4  SortedList results=∅, batch=∅;
5  CurrentRelation cr = A;
6  while (true) {
7    if (cr == A) {
8      myTuples = tuplesA.get(kv.joinValue);
9      otherTuples = tuplesB.get(kv.joinValue);
10   } else {
11     myTuples = tuplesB.get(kv.joinValue);
12     otherTuples = tuplesA.get(kv.joinValue);
13   }
14   batch.insert(next C_cr rows from CF "cr");
15   foreach(row ∈ batch) {
16     foreach(kv ∈ row) {
17       myTuples.append(kv.joinValue, kv);
18       foreach (kv' ∈ otherTuples) {
19         results.add(innerJoin(kv, kv'));
20         if (HRJNTerminationTest(results,
               tuplesA, tuplesB) == true)
21           return (results);
22       }
23     }
24   }
25   batch.clear();
26   cr = (cr == A) ? B : A;
27 }
28 return (results);
```

the current threshold value, and terminates when it is below the score of the k'th tuple in the result set.

5. STATISTICAL RANK-JOINS

Both of the previous algorithms, ship tuples even though they may not participate in the top-k result set. Our next contribution aims to avoid this. Note that we need not only estimate which tuples will produce the join result, but also to predict whether these tuples can have a top-k score.

5.1 The BFHM Data Structure

The BFHM index is a two-level statistical data structure, encompassing histograms and Bloom filters. At the first level, we have an equi-width histogram on the score axis; that is, all histogram buckets have the same spread and each such bucket stores information for tuples whose scores lie within the boundaries of the bucket. At the second level, instead of a simple counter per bucket (plus the actual min and max scores of tuples recorded in the bucket), we choose to maintain a Bloom filter-like data structure, recording the join values of the tuples belonging to the bucket. This will then be used to estimate the cardinality of the join result set during query processing, to be discussed shortly. In brief, the BFHM data structure has two main parameters: the number of buckets in the BFHM (*numBuckets*), and the number of bits in each BFHM bucket Bloom filter (m).

As false positives can inflate the join cardinality estimation, we have opted for a fusion scheme, combining single-hash-function Bloom filters with Counting Bloom filters and compression. More specifically, in each BFHM bucket we maintain: (i) the minimum and maximum score values of tuples recorded in the bucket; (ii) a single-hash-function

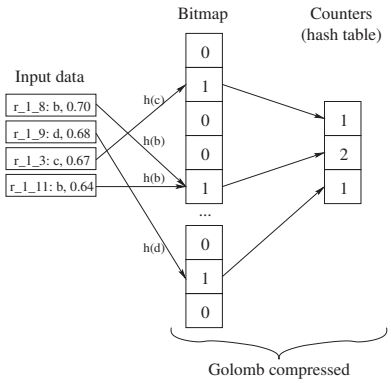


Figure 4: BFHM bucket structure

Algorithm 5 BFHM Index Creation

```
1  Input: Rows from a single column family (e.g., A), of
       the form {row.rowKey: row.joinValue, row.score},
       number of buckets in BFHM (numBuckets)
2  Output: BFHM blob rows and reverse mappings for A
3  Map():
4    foreach(row ∈ sequential scan of A) {
5      int bucketNo = scoreToBucket(row.score,
           numBuckets);
6      emit(bucketNo: {row.rowKey:
           row.joinvalue, row.score});
7    }
8  Reduce():
9    HybridBloomFilter filter; // see sec. 5.1
10   float minScore = ∞, maxScore = -∞;
11   foreach(kv ∈ input) {
12     int bitPos = filter.insert(row.joinValue);
13     if (row.score < minScore)
14       minScore = row.score;
15     if (row.score > maxScore)
16       maxScore = row.score;
17     emit(bucketNo|bitPos: {row.rowKey:
           row.joinvalue, row.score});
18   }
19   emit(bucketNo, {GolombCompress(filter),
         minScore, maxScore});
```

Bloom filter of size m (bits); and (iii) a hash table of counters for each non-zero bit of the Bloom filter. Both of the latter two constructs are then compressed using Golomb coding[11]. The resulting data structure is a hybrid between Golomb Compressed Sets and Counting Bloom filters, allowing us at the same time to (i) minimize the false positive probability for our treatment of Bloom filters for join cardinality estimation (to be discussed shortly), (ii) greatly reduce the amount of bytes stored in the NoSQL store and transferred across the network, and (iii) achieve a reasonable trade-off between compression ratio and processing costs. Fig. 4 depicts a pictorial of how data are inserted to the Bloom filter-related section of the BFHM bucket for the score range $(0.60, 0.70]$ for tuples of relation R_1 in our running example. Please note that the compression of the bit vector and counter hash table is an integral part of our data structure, as single hash function Bloom filters can grow very large in space and are thus impractical otherwise. Moreover, to our knowledge, this is the first work to propose, implement, and evaluate such a fusion scheme.

Like with our indices, the BFHM data are stored in the

Row key	Index tuples ([*blob*],score$_{min}$,score$_{max}$ or {row key: join value, score})	
	R_1	R_2
0	[*blob*],0.93,1.00	[*blob*],0.91,0.92
0|$h(a)$	{r$_{1_{10}}$: a,1.00}	
0|$h(b)$		{r$_{2_2}$: b,0.91},{r$_{2_{11}}$: b,0.92}
0|$h(c)$	{r$_{1_2}$: c,0.93}	
1	[*blob*],0.82,0.82	
1|$h(b)$	{r$_{1_7}$: b,0.82}	
1|$h(d)$	{r$_{1_1}$: d,0.82}, {r$_{1_4}$: d,0.82}	
2	[*blob*],0.70,0.79	
2|$h(a)$	{r$_{1_5}$: a,0.73}	
2|$h(b)$	{r$_{1_8}$: b,0.70}	
2|$h(c)$	{r$_{1_6}$: c,0.79}	
3	[*blob*],0.64,0.68	[*blob*],0.64,0.64
3|$h(b)$	{r$_{1_{11}}$: b,0.64}	
3|$h(c)$	{r$_{1_3}$: c,0.67}	{r$_{2_3}$: c,0.64}
3|$h(d)$	{r$_{1_9}$: d,0.68}	
4		[*blob*],0.50,0.53
4|$h(a)$		{r$_{2_1}$: a,0.51}
4|$h(d)$		{r$_{2_4}$: d,0.53}, {r$_{2_6}$: d,0.50}
5		[*blob*],0.41,0.41
5|$h(d)$		{r$_{2_5}$: d,0.41}
6		[*blob*],0.31,0.38
6|$h(a)$		{r$_{2_7}$: a,0.35}, {r$_{2_8}$: a,0.38}, {r$_{2_9}$: a,0.37}
6|$h(c)$		{r$_{2_{10}}$: c,0.31}

Figure 5: Running example: BFHM index table

NoSQL store. More specifically, for each input relation, the BFHM index is stored in a separate column family or index table. Each BFHM bucket is stored in a separate row with the bucket number as its key (e.g., for scores in $[0, 1]$ and 10 buckets, the first bucket – i.e., for score values in $(0.9, 1.0]$ – will be stored under key 0, the bucket for score values in $(0.8, 0.9]$ will use key 1, and so on); the row values then include the min and max actual scores, plus the Golomb-compressed bitmap and counters' hashtable (coined BFHM bucket "blob"). Moreover, as we shall see shortly, during query processing we need to be able to map BFHM set bit positions back to the corresponding join values. As this is not possible with most quasi-random hash functions, we need to further store these mappings. These are stored in the same column family/table as the above data, in rows where the key consists of the concatenation of the bucket number and bit position, and where the row data includes tuples of the form $\{rowkey : join\ value, score\}$. Assuming that $h(x)$ is the bit position indicated for item x by the hash function used by our Bloom filter, Fig. 5 depicts the BFHM table contents for our running example. This is created with a MapReduce job (Algorithm 5). In the Map phase, the mappers partition incoming tuples into the various histogram buckets. Each reducer operates on the mapped tuples for one BFHM bucket at a time. Each incoming tuple is first added to the BFHM hybrid filter based on its join value, and its corresponding bit position is recorded. The reducer emits a reverse mapping entry for each such tuple, and keeps track of the min and max scores of all tuples in the bucket. When the bucket tuples are exhausted, the reducer finally emits the BFHM bucket blob row.

5.2 BFHM Query Processing

Query processing consists of two phases: (i) estimating the result, and (ii) reverse mapping and computation of the true result. Algorithm 6 shows the 1st phase. The "coordinator" fetches BFHM bucket rows for the joined relations, one at a time, with newly fetched buckets being "joined" with older ones. The bucket join result – an estimation of the join result for tuples recorded in the joined buckets – is then added to the list of estimated results. When the estimated number of result tuples in this list (i.e., the sum of cardinalities of added buckets) is above k, the algorithm tests for the BFHM termination condition, to be discussed shortly. If the latter is satisfied, processing continues with the reverse mapping/final result set computation phase.

Algorithm 6 BFHM Rank-Join Estimation

```
1  Input: Rows from the BFHM index for A and B (Fig. 5)
2  Output: Estimated top-k join result set
3  List bucketsA=∅, bucketsB=∅, myBuckets,
       otherBuckets;
4  SortedList results=∅; // Sorted on maxScore
5  int numEstimatedResults = 0;
6  CurrentRelation cr = A;
7  BFHM newBucket;
8  boolean done = false;
9  while (!done) {
10   if (cr == A) {
11     myBuckets = bucketsA;
12     otherBuckets = bucketsB;
13   } else {
14     myBuckets = bucketsB;
15     otherBuckets = bucketsA;
16   }
17   newBucket = fetchNextBucketFrom(BFHM_{cr});
18   myBuckets.add(newBucket);
19   foreach(bucket ∈ otherBuckets) {
20     EstimatedResult res = bucketJoin(newBucket,
           bucket); // See alg. 7
21     if (res == null)
22       continue;
23     results.add(res);
24     numEstimatedResults += res.cardinality;
25     if (BFHMTerminationTest(bucketsA,bucketsB,
           numEstimatedResults)) {
26       done = true;
27       break;
28     }
29   }
30   cr = (cr == A) ? B : A;
31 }
32 return (results);
```

Algorithm 7 outlines the bucket join procedure. First, we compute the bitwise-AND of the Bloom filter bitmaps from the two buckets; if the resulting bitmap is empty (i.e., all bits are 0), then there are no joining tuples recorded in these two buckets. Otherwise, we compute an estimation of the cardinality of the join, by summing up the products of the counters corresponding to the non-zero bit positions in the result filter. The factor α (line 9) is there to compensate for false positives in the filters; for now, assume $\alpha = 1$. Last, we compute the min and max score of any join result tuple from these buckets, by using the actual min and max scores of the joined buckets as input to the aggregate score function.

Fig. 6(c) shows the estimated result set for our running example, using *sum* as the aggregate scoring function; the join attribute value is shown as $h(\cdot)$ to denote that we refer to non-zero positions in the bitwise-AND of filter bitmaps and not to actual join values, while the Bloom filter counters are given in consolidated form in Fig. 6(a) and 6(b) for clarity. First, the algorithm would fetch the $(0.9, 1.0]$ buckets for R_1 and R_2; their bucket-join would return *null* as they have no common non-zero bit position. The algorithm

Algorithm 7 BFHM bucket join

```
1  Input: BFHM_{R1}[i] (i'th bucket from R1's
       BFHM), BFHM_{R2}[j] (j'th bucket from R2's BFHM)
2  Output: join result estimation for this bucket pair
3  EstimatedResult res;
4  res.BF = BFHM_{R1}[i].BF & BFHM_{R2}[j].BF;
5  if (res.BF == ∅)
6      return null;
7  foreach(bit ∈ res.BF non-zero bits)
8      res.cardinality += BFHM_{R1}[i].counters(bit) *
           BFHM_{R2}[j].counters(bit) * α;
9  res.minScore = joinScore(BFHM_{R1}[i].minScore,
       BFHM_{R2}[j].minScore);
10 res.maxScore = joinScore(BFHM_{R1}[i].maxScore,
       BFHM_{R2}[j].maxScore);
11 return res;
```

	0.9 – 1.0	0.8 – 0.9	0.7 – 0.8	0.6 – 0.7
min	0.93	0.82	0.70	0.64
max	1.00	0.82	0.79	0.68
h(a)	1		1	
h(b)		1	1	1
h(c)	1		1	1
h(d)		2		1

(a) R_1 BFHM

	0.9 – 1.0	0.6 – 0.7	0.5 – 0.6	0.4 – 0.5	0.3 – 0.4
min	0.91	0.64	0.50	0.41	0.31
max	0.92	0.64	0.53	0.41	0.38
h(a)			1		3
h(b)	2				
h(c)		1			1
h(d)			2	1	

(b) R_2 BFHM

#	Join Attr	Min Score	Max Score	# of est. Results	R_1 bucket	R_2 bucket
1	h(b)	1.73	1.74	2	0.8–0.9	0.9–1.0
2	h(b)	1.61	1.71	2	0.7–0.8	0.9–1.0
3	h(c)	1.57	1.64	1	0.9–1.0	0.6–0.7
4	h(b)	1.55	1.60	2	0.6–0.7	0.9–1.0
5	h(a)	1.43	1.53	1	0.9–1.0	0.5–0.6
6	h(c)	1.34	1.43	1	0.7–0.8	0.6–0.7
7	h(d)	1.32	1.35	4	0.8–0.9	0.5–0.6
8	h(c)	1.28	1.32	1	0.6–0.7	0.6–0.7
9	h(a)	1.24	1.38	3	0.9–1.0	0.3–0.4
10	h(c)	1.24	1.38	1	0.9–1.0	0.3–0.4
11	h(d)	1.23	1.23	2	0.8–0.9	0.4–0.5
12	h(a)	1.20	1.32	1	0.7–0.8	0.5–0.6
13	h(d)	1.14	1.21	2	0.6–0.7	0.5–0.6
14	h(d)	1.05	1.09	1	0.6–0.7	0.4–0.5
15	h(a)	1.01	1.17	3	0.7–0.8	0.3–0.4
16	h(c)	1.01	1.17	1	0.7–0.8	0.3–0.4
17	h(c)	0.95	1.06	1	0.6–0.7	0.3–0.4

(c) Estimated BFHM join result (score function: *sum*)

Figure 6: Example: BFHM join result estimation

would then proceed by fetching bucket $(0.8, 0.9]$ for R_1 and joining it to bucket $(0.9, 1.0]$ for R_2; the join would return an estimated result containing two tuples (the product of the counters for bit position $h(b)$), with a minimum score of $0.82+0.91 = 1.73$ and a maximum score of $0.82+0.92 = 1.74$. Then it would be R_2's turn, fetching the $(0.6, 0.7]$ bucket and joining it to the two buckets already fetched for R_1, etc.

To test for the termination of the estimation phase, we examine the estimated results list and the buckets fetched so far. First, we compute the minimum score of the k'th estimated result. The estimation phase terminates if there are more than k estimated results and there is no combination of buckets not examined so far that could have a maximum score above that of the k'th estimated result.

Take for example the estimated result of Fig. 6(c) and assume we requested the top-3 join results. After having fetched the first two buckets for R_1 (i.e., $(0.9, 1.0]$, $(0.8, 0.9]$) and for R_2 (i.e., $(0.9, 1.0]$ and $(0.6, 0.7]$), we would have computed rows 1 and 3 of the result set in Fig. 6(c). At this time, the estimated result set consists of $2 + 1 = 3$ estimated tuples, and the minimum score of the third tuple would be 1.57. The maximum attainable score for the join of the next bucket (i.e., bucket $(0.7, 0.8]$) of R_1 and the highest-score bucket of R_2 (i.e., $(0.9, 1.0]$) would be $0.8 + 1.0 = 1.8$ which is higher than 1.57 so the estimation phase does not terminate. After fetching and joining bucket $(0.7, 0.8]$ of R_1, the result set would consist of rows 1, 2, 3, and 6 of Fig. 6(c). Now the estimated score for the top-third result becomes 1.71. The maximum attainable score for the join of the next bucket of R_2 (bucket $(0.5, 0.6]$) against the highest-scoring bucket of R_1 would be 1.6; conversely, the maximum attainable score for the next bucket of R_1 (bucket $(0.6, 0.7]$) against bucket $(0.9, 1.0]$ of R_2 is 1.7; since both of these are lower than 1.71, the estimation phase terminates.

The next phase examines the estimated results of the first phase and purges all estimated results whose maximum score is below that of the (estimated) k'th tuple. Then, the algorithm fetches the reverse mapping rows corresponding to the non-zero bit positions of the Bloom filters in the estimated results, which are then used to compute the final result set.

5.3 Analysis of BFHM Rank-Join

Our BFHM-based algorithms deal with two sources of inaccuracy when estimating the rank join result set: use of histograms and use of Bloom Filters. The former introduces errors in the estimation of the actual score of the join results, while false positives in the latter may result in overestimating the join result size. For ease of presentation, assume for now that our Bloom filters are false-positive free. This allows us to know for sure when joining tuples from any two given buckets of the BFHM will actually produce join results. However, it gives us no way of knowing the exact scores of the joined tuples and thus does not allow us to compute the actual score of the join result tuple.

In order to accomplish this, we maintain the min and max score achievable when joining two buckets; then, instead of keeping the k highest scored estimated results, our algorithms also keep all those tuples whose maximum possible score is larger than the lowest possible score of the k^{th} estimated result. This guarantees that no tuple is lost from the final result set, at the expense of fetching/storing some tuples that may not make it in the final result set.

In a false-positive-free world, this would suffice. Alas, false positives in the Bloom filters of the BFHM may cause an overestimation of the join result set size for any two joined buckets. Surely one can tweak the Bloom filter parameters so as to minimize the false positive probability; however, doing this may lead to overly large Bloom filters, thus diminishing any bandwidth consumption returns expected from their use. Moreover, even if very large BFs were practical, one can not guarantee that no false positives arise, as a result for example of a BFHM bucket being overpopulated. In order to deal with this, we incorporate the effective false positive probabilities of the Bloom filters in the join result size estimation (the α factor in algorithm 7). Given a single-hash-function Bloom filter of size m, after having inserted n distinct items, the probability that a given bit is set equals $P_T = (1 - (1 - 1/m)^{kn}) \approx 1 - e^{-m/kn}$. In the case of counting Bloom filters, we can use this information to estimate a "compensated" value of any given counter, as follows. When joining two filters (say BF_A and BF_B), and $JSize$ is the estimated join size as computed through $BF_A \cdot BF_B$ (i.e., sum

of products of matching Bloom filter counter values), we scale $JSize$ by a factor of $\alpha = (1 - P_{T_A}) \cdot (1 - P_{T_b})$; that is:

$$JSize_{A \cdot B} = BF_A \cdot BF_B \cdot (1 - P_{T_A}) \cdot (1 - P_{T_b})$$

Our experimental evaluation showed that the combination of these two mechanisms results in a 100% recall for all workloads and parameter values tested. Apart from the above probabilistic scheme, we can further *guarantee* a 100% recall, as follows. First, when we have k or more results in the final result set, we examine the score of the k^{th} actual join result and compare it to the maximum scores of BFHM buckets that didn't make it to the fetch list. If there are buckets whose maximum score is above the former, then we should consider these additional buckets too. If no change occurs in the result set after this step, the algorithm terminates. Similarly, for the case were $k' < k$ results have been produced in the second query processing phase, we resume the query processing algorithm from the point it initially stopped, only now looking for the top-$k + (k - k')$ results. When k or more results have been produced, the algorithm performs the checks outlined in the first case.

LEMMA 1. *The set of tuples represented by the BFHM resulting by combining the BFHMs of multiple joined buckets, is a superset of the actual join result set.*

PROOF. Remember that tuples are inserted to the BFHM based on their join attribute value, and that when joining two (or more) such structures, we first perform a bitwise-AND of the Bloom filters. This means that, in the resulting BFHM, bit positions that were unset in at least one of the BFHMs, will also be 0, while all remaining positions (i.e., for which all BFHMs had a non-zero value) will be non zero, and the product of the respective counters will give us an estimation of their join result size. Being based on Bloom filters, each individual BFHM cell can only introduce false positives; that is, in our context, the individual counters in every counter position of the original BFHMs will be equal to or larger than the cardinality of the values they represent. Hence, we can only overestimate the number of join results corresponding to any position in the final BFHM. □

In essence, this means that the recall of our BFHM-based algorithm is not affected by the use of Bloom filters. It thus suffices to prove that our treatment of BFHM cells/buckets is such that if some item is missing from the output, then our algorithms can detect and fetch it.

THEOREM 1. *The BFHM-based rank join algorithms can achieve a 100% recall for any valid input.*

PROOF. We shall prove this by contradiction. Let t be a join result tuple which should be in the top-k join result set but is omitted by our algorithm; that is, t's score is among the actual top-k join result scores but t is not in the final result set computed during phase 2 (and possible repetitions, as discussed above). Given lemma 1, this may happen only if the algorithm has stopped before examining the join result BFHM bucket in which t belongs. This in turn means that the result set consists of at least k results, and that the maximum score of t's bucket (being larger or equal to t's score) – and hence t's score – is below the score of the k^{th} result; a contradiction. □

6. UPDATES AND MAINTENANCE

Both the IJLMR and ISL indexes are in essence space-optimized inverted lists of the base data. To maintain these indexes up-to-date in the face of concurrent run-time updates, we have overloaded the base data insertion and deletion functions, intercepting these primitives so as to also propagate changes to the index. More specifically, both insertions and deletions are intercepted at the caller level; then, the mutation is augmented so as to perform both a base data *and* an index insertion/deletion in one operation, using the original mutation timestamp for both operations. This reduces the time between data and index updates, and takes a step towards index consistency. We have opted for eventual consistency, since this is the consistency level also natively supported by most contemporary NoSQL stores; failed mutations are retried until successful and key-value timestamps are used to discern between fresh and stale tuples.

For BFHM, the existence of the BFHM blobs makes concurrent updates more complicated. To this end, we have taken a hybrid approach, where updates to the reverse mappings are performed just as above, while updates to the blobs are handled through special *insertion* and *tombstone* records. These are key-value pairs that are stored in the bucket row, along with the blob and bucket score range. Each tuple insertion in a specific BFHM bucket will result in an "insertion" record being added to the bucket row (in addition to an entry being added in the corresponding reverse mapping row); this key-value pair holds all BFHM-related information (i.e., the tuple's rowkey, join value, and score), and bears the same timestamp as the newly inserted tuple. Conversely, if a tuple is deleted from a BFHM bucket, then a "tombstone" record is added to the bucket row, again bearing all BFHM-related information and the same timestamp as the delete operation; reverse mappings are directly deleted, using the NoSQL store's vanilla delete operation. This information allows anyone retrieving a bucket row to replay all row mutations in timestamp order and reconstruct the up-to-date blob from the original blob. The blob is then written back to the NoSQL store using the timestamp of the latest replayed mutation, and insertion/tombstone records with an older or equal timestamp are purged, all in a single operation. HBase (and most NoSQL stores) support row-level atomicity; coupled with the above treatment of timestamps, this ensures that no updates are lost. The blob write-back can be performed *eagerly* (at the beginning of query processing), *lazily* (after the query results are returned to the user), or *off-line* (by a thread periodically probing bucket rows for mutation records). Moreover, one can choose to perform the write-back only if the number of replayed mutations is above some predefined threshold.

7. EXPERIMENTAL EVALUATION

7.1 Methodology

We implemented all of the above mentioned algorithms, comprising approx. 6k lines of Java code. We further implemented the DRJN algorithm from [8]. The DRJN index is roughly a 2-d matrix, with join value partitions on its x-axis and score value partitions on its y-axis. DRJN query processing proceeds as follows: (i) the querying node fetches complete DRJN matrix rows in decreasing score order; (ii) the relevant buckets are "joined" so as to estimate the cardinality of their join; (iii) when the cumulative cardinality

surpasses k, contact all nodes and fetch and join all tuples whose score is above the the lower score boundaries of the last fetched buckets; (iv) terminate if the cardinality of the actual result set is k and the score of the k'th tuple is larger than the maximum attainable join score of the last fetched buckets, otherwise loop to (i), incrementally fetching more buckets and tuples. As [8] was designed for a P2P-like setting, we had to revisit it so as to be useable in a NoSQL store such as HBase. First, we opted to group DRJN buckets by their scores and store all buckets for a given score range as columns of a single row; thus, the querying node will retrieve a complete batch of buckets, as required at step (i) above, with a single HBase *Get()* operation. We further augmented HBase with custom server-side filters to allow for efficient filtering of tuples in step (iv). Last, we further expedited step (iv) by implementing it as a lightweight Map-only Hadoop job, storing its output data in a temporary HBase table for the querying node to access and join.

We employed two different clusters: one in a controlled lab environment and one "in-the-wild". For the latter, we used Amazon's Elastic Compute Cloud (EC2), with clusters consisting of 3, 5, and 9 m1.large nodes. (each with 2 virtual cores, 7.5 GB RAM, and 2x 420 GB of instance storage). The lab cluster (LC) consisted of 5 nodes, each with 2 CPUs, 16 cores per CPU, 64GB RAM, and 10× 1TB disks.

We used the TPC-H generator, generating data for the "Lineitem", "Orders", and "Part" tables, for scale factors from 10 to 500. The larger (smaller) data scale resulted in tables with 3 billion (60M), 750 million (15M), and 100 million (2M) rows, which occupied ≈1.7TB (34GB), ≈200GB (4GB), and ≈25GB (0.5GB) of HBase disk space, respectively. With the TPC-H generator we also computed update sets, to be applied to the base data and indexes. All Bloom filters were configured to contain the most heavily populated of the buckets with a false positive probability of 5%. The number of BFHM buckets was set to 100 and 1000 on EC2 and to 100 and 500 on LC. ISL was configured with batching sizes matching the number of BFHM buckets; 1% and 0.1% on EC2, and 1% and 0.2% on LC. DRJN was also configured with 100 and 500 buckets on LC.

We used the following queries:

```
Q1: SELECT * FROM Part P, Lineitem L
    WHERE P.PartKey=L.PartKey
    ORDER BY (P.RetailPrice * L.ExtendedPrice)
    STOP AFTER k
Q2: SELECT * FROM Orders O, Lineitem L
    WHERE O.OrderKey=L.OrderKey
    ORDER BY (O.TotalPrice + L.ExtendedPrice)
    STOP AFTER k
```

These queries were selected to showcase both the use of different aggregate scoring functions and the effect of score value distributions on the query processing time. Queries were executed 20 times and we report on the average values (the standard deviation was too small to show on the graphs in all cases). Last, we applied the update sets one at a time and executed 20 repetitions of the same queries. All block-level/memory caches were purged between consecutive update and query executions.

We evaluate all algorithms using the following metrics:

- Turnaround time: the wall-clock time required to compute and return the top-k join result.
- Network bandwidth: the number of bytes transferred through the network.
- Dollar cost: the number of tuples read from the cloud store during query processing[1].

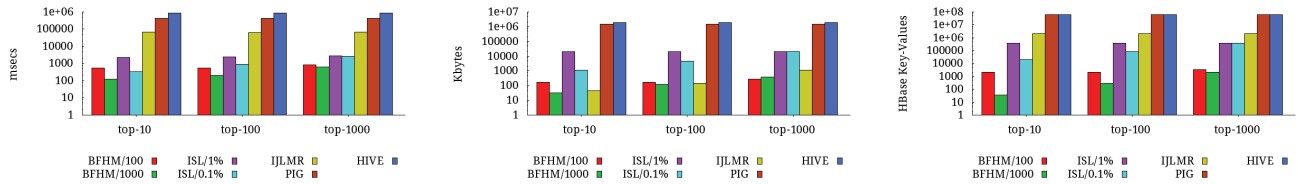
Regarding query processing times, the BFHM and ISL algorithms were similar across all EC2 cluster sizes. For the PIG, HIVE, and IJLMR approaches, the increase in cluster size resulted in a ≈30% decrease in processing time going from 1+2 to 1+8 nodes, with the rest of the metrics being roughly the same across cluster sizes. Thus, to save space, we present only the figures for the 1+8 EC2 cluster and a scalefactor of 10 (denoted "EC2"), and for the 5-node lab cluster and a scalefactor of 500 ("LC"). With respect to query response time, IJLMR, PIG, and HIVE had significantly reduced performance. Specifically, IJLMR, was consistently worse than the next-best algorithm by up to an order of magnitude, PIG was worse than IJMLR by about an order of magnitude, and HIVE was worse than PIG by about an order of magnitude. Thus, for presentation clarity we omit specific results for these when showing the LC results with the big scale factor.

7.2 Results

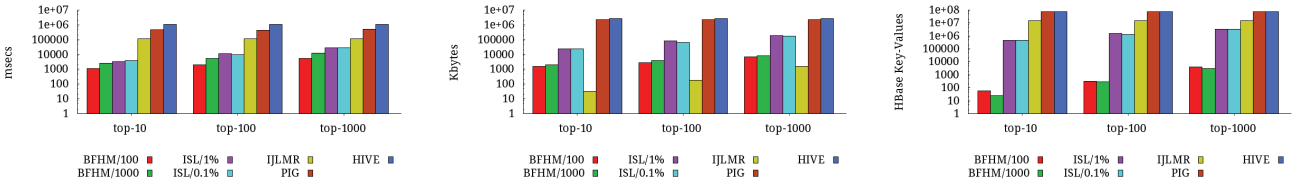
Query Processing Time. Figures 7(a), 7(d), 8(a), and 8(d) depict the time required by each algorithm to process the Q1 and Q2 top-k join queries, for various values of k. Please note the logarithmic y-axis. Contrasting the results for Q1 and Q2 we can see how the different score distributions affect the processing time. For Q2 there are fewer high-ranking tuples, thus we need to reach deeper into each index to produce the top-k result set compared to Q1. On EC2, BFHM is the clear winner across the board, with ISL following, and IJLMR, PIG, and HIVE trailing by large margins. For LC, ISL is shown to be best, with BFHM closing the gap and occasionally beating ISL, as k increases. DRJN trails by several orders of magnitude, primarily due to the cost of the Map jobs needing to scan the whole dataset to send to the coordinator those rows having a score greater that the threshold calculated by it.

Query Processing Bandwidth Consumption. Figures 7(b), 7(e), 8(b), and 8(e) depict the bandwidth consumed to process Q1 and Q2. IJLMR does in general very well, as it only transfers the local top-k lists from the mappers to the sole reducer. However, as k increases, BFHM closes the gap, eventually even winning for large values of k. In general, DRJN achieves its best performance for this metric. From the LC results, DRJN is the clear winner for Q1 and for low-k values for Q2. Note that in DRJN, although its mappers need to scan the complete dataset, this is typically fetched from each mapper's local disk. Further, our optimization of server-side filtering paid off, as the amount of data put on the network is significantly reduced. For Q1, where the top-k join is computed very early, DRJN shines, as very little data need be fetched over the network. For the more demanding Q2 however, as k increases, its improvement over BFHM becomes much smaller.

[1]Per DynamoDB's pricing scheme[9], each key-value read from the NoSQL store corresponds to 1 unit of *Read Capacity* (as all of our key-value pairs are less than 1KB in size), with *Read Throughput* being priced at $0.01 per hour for every 50 units of Read Capacity.

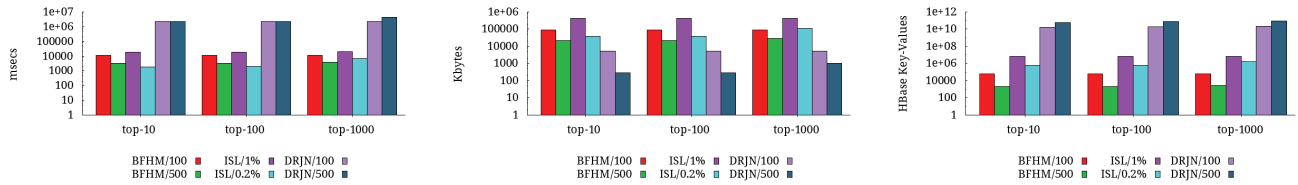


(a) Q1: Query processing time (b) Q1: Network bandwidth (c) Q1: Dollar cost

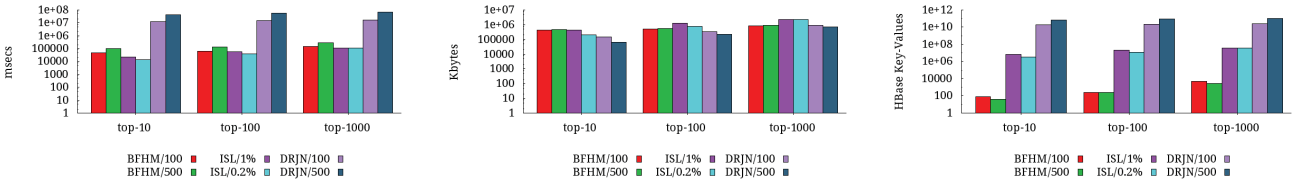


(d) Q2: Query processing time (e) Q2: Network bandwidth (f) Q2: Dollar cost

Figure 7: Results for Q1 and Q2 on EC2



(a) Q1: Query processing time (b) Q1: Network bandwidth (c) Q1: Dollar cost



(d) Q2: Query processing time (e) Q2: Network bandwidth (f) Q2: Dollar cost

Figure 8: Results for Q1 and Q2 on LC

Query Processing Dollar-cost. Following the DynamoDB pricing model, Figures 7(c), 7(f), 8(c), and 8(f) depict the number of key-value pairs read from the NoSQL store. Naturally, the MapReduce approaches are the worst, since they need to scan all of the input data. BFHM, with its accuracy in estimating the result set cardinality, and its "surgical" accuracy in retrieving appropriate tuples from the input relation, is the clear winner here with ≈1-3 orders of magnitude less cost than the next best contender (ISL) and up to 5 orders of magnitude better than DRJN.

Indexing Costs. Fig. 9 depicts the indexing times, showing that our indexing algorithms scale well with the cluster and dataset sizes. We stress that, across the board, the sum of the index building time plus the relevant query processing times shown earlier, is on par or lower than the time required to execute the same query in PIG (and much faster than executing it in HIVE). In essence, this means that we can afford to build our indices just before executing a query, and still be competitive against PIG or HIVE! Additionally, we report on the storage space used by each index and the maximum memory footprint of individual mappers/reducers during the index building stages, on the Lab Cluster and for the 500-scalefactor. More specifically, the disk space used by each index (for Part, Orders, and Lineitem resp.) was:

- BFHM: 2.6, 22, and 110 GB (incl. reverse mappings)
- ISL: 1.2, 13.5, and 85 GB
- IJLMR: 1.2, 13.5, and 85 GB
- DRJN: ranging from 400 kB (100 buckets) to 8.5 MB (500 buckets)

Keep in mind that the on-disk size of the base relations was 25 GB, 200 GB, and 1.7 TB respectively for Part, Orders, and Lineitem. The memory footprint of reducers during the index building phase was:

- BFHM/100 buckets: 4 GB worst case, 1 GB average.
- BFHM/500 buckets: 2 GB worst case, 0.5 GB average
- ISL/IJLMR: negligible
- DRJN: ranging from 3.5 MB (Part, 500 buckets) to 125 MB (Lineitem, 100 buckets)

Online Updates. Last, we studied the effect of online updates for BFHM. We first used the TPC-H generator to generate a number of update sets, each consisting of $\approx s \times 600$ insertions and $\approx s \times 150$ deletions for scale-factor s. We then applied each of these sets in their entirety (i.e., ≈ 750 mutations), followed by a single query for which we measured the query processing time. Even with the "eager" update scheme (i.e., the coordinator reconstructed and wrote back the updated BFHM at the beginning of query processing), fitting an update-heavy workload – a worst-case scenario

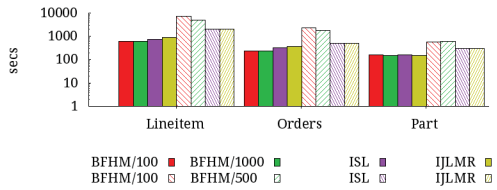


Figure 9: Indexing time (solid: EC2; pattern: LC)

with regard to the query processing time overhead – the overall time overhead was less than 10% across the board (figure omitted due to space reasons).

8. CONCLUSIONS

Top-k join queries arise naturally in many real-world settings. We studied algorithms for rank joins in NoSQL stores. This is, to our knowledge the first such endeavor. We contributed novel algorithms, implemented them, and extensively tested their performance over Amazon EC2 and in-house clusters, using TPC-H data at various scales and different query types. The central conclusion is that for all metrics, data sets, and query types studied, the BFHM Rank Join algorithm is very desirable. It typically manages to outperform the others and even when it is not the best approach, it offers a performance that is in absolute terms satisfactory and is less sensitive to the various query types, data sets, k values, and their configuration parameters. Immediate future plans include the adoption of dynamic Bloom filters to further improve the time and bandwidth performance of BFHM Rank Join, as well as an exploration of the design space with regard to our various system parameters.

Acknowledgements. This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

9. REFERENCES

[1] A. Abouzeid, et al. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[2] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *Proc. EDBT*, 2010.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426.

[4] C. Böhm and H.-P. Kriegel. A cost model and index architecture for the similarity join. In *Proc. ICDE*, 2001.

[5] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *Proc. ACM PODC*, 2004.

[6] S. Cohen and Y. Matias. Spectral Bloom filters. In *Proc. ACM SIGMOD*, 2003.

[7] J. Dittrich, et al. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3(1-2):515–529, 2010.

[8] C. Doulkeridis, et al. Processing of rank joins in highly distributed systems. In *IEEE ICDE*, 2012.

[9] DynamoDB pricing scheme: `http://aws.amazon.com/dynamodb/#pricing`.

[10] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. ACM PODS*, 2001.

[11] S. W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12(3):399, 1966.

[12] I. Ilyas, W. Aref, and A. Elmagarmid. Joining ranked inputs in practice. In *Proc. VLDB*, 2002.

[13] I. Ilyas, W. Aref, and A. Elmagarmid. Supporting top-k join queries in relational databases. In *Proc. VLDB*, 2003.

[14] I. Ilyas, G. Beskales, and M. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4):1–58, 2008.

[15] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu. Llama: leveraging columnar storage for scalable join processing in the mapreduce framework. In *Proc. ACM SIGMOD*, 2011.

[16] S. Michel, P. Triantafillou, and G. Weikum. KLEE: A framework for distributed top-k query algorithms. In *Proc. VLDB*, 2005.

[17] M. Mitzenmacher. Compressed Bloom filters. *IEEE/ACM Transactions on Networking*, 10(5):604–612, 2002.

[18] J. Mullin. Estimating the size of a relational join. *Information Systems*, 18(3):189–196, 1993.

[19] A. Natsev, Y.-C. Chang, J. Smith, C.-S. Li, and J. Vitter. Supporting incremental join queries on ranked inputs. In *Proc. VLDB*, 2001.

[20] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *Proc. ACM SIGMOD*, 2011.

[21] C. Olston, et al. Pig Latin: A not-so-foreign language for data processing. In *Proc. ACM SIGMOD*, 2008.

[22] K. Schnaitter and N. Polyzotis. Evaluating rank joins with optimal cost. In *Proc. ACM PODS*, 2008.

[23] M. Stonebraker, et al. Mapreduce and parallel DBMSs: Friends or foes? *Comm. ACM*, 53(1):64–71, 2010.

[24] A. Thusoo, et al. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[25] M. Wu, L. Berti-Equille, A. Marian, C. Procopiuc, and D. Srivastava. Processing top-k join queries. *PVLDB*, 3(1-2):860–870, 2010.

[26] C. Xia, H. Lu, B. C. Ooi, and J. Hu. Gorder: An efficient method for kNN join processing. In *Proc. VLDB*, 2004.

[27] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *Proc. ICDE*, 2009.

[28] D. Zeinalipour-Yazti, et al. The Threshold Join Algorithm for top-k queries in distributed sensor networks. In *Proc. ACM DMSN*, 2005.

[29] K. Zhao, S. Zhou, K.-L. Tan, and A. Zhou. Supporting ranked join in peer-to-peer networks. In *Proc. DEXA*, 2005.

Crowdsourcing Taxonomies*

Dimitris Karampinas and Peter Triantafillou

Computer Engineering & Informatics Department, University of Patras
{karampin,peter}@ceid.upatras.gr

Abstract. Taxonomies are great for organizing and searching web content. As such, many popular classes of web applications, utilize them. However, their manual generation and maintenance by experts is a time-costly procedure, resulting in static taxonomies. On the other hand, mining and statistical approaches may produce low quality taxonomies. We thus propose a drastically new approach, based on the proven, increased human involvement and desire to tag/annotate web content. We define the required input from humans in the form of explicit structural, e.g., supertype-subtype relationships between concepts. Hence we harvest, via common annotation practices, the collective wisdom of users with respect to the (categorization of) web content they share and access. We further define the principles upon which crowdsourced taxonomy construction algorithms should be based. The resulting problem is NP-Hard. We thus provide and analyze heuristic algorithms that aggregate human input and resolve conflicts. We evaluate our approach with synthetic and real-world crowdsourcing experiments and on a real-world taxonomy.

Keywords: Collective Intelligence, Crowdsourcing, Taxonomy, Tagging

1 Introduction

Social media applications and research are increasingly receiving greater attention. A key defining characteristic is the increased human involvement. Even before todays' success of social media applications, many applications became extremely successful due to the clever exploitation of implicit human inputs (e.g., Google's ranking function), or explicit human input (e.g., Linux open source contributions). Social media and the social web have taken this to the next level. Humans contribute content and share, annotate, tag, rank, and evaluate content. Specialized software aggregates such human input for various applications (from content searching engines to recommendation systems, etc). The next wave in this thread comes from *crowdsourcing systems* in which key tasks are performed by humans (either in isolation or in conjunction with automata) [7]. Lately, within the realm of data and information retrieval systems, crowdsourcing is gaining momentum as a means to improve system performance/quality [3,

* This research is partially funded by the EIKOS research project, within the THALES framework, administered by the General Secretariat of Research and Technology, Greece.

13]. A recent contribution suggests to engage humans during the processing of queries for which humans are better suited for the task (e.g., entity disambiguation) [8]. Further, (anthropocentric) data systems are proposed whose functioning (including semantics enrichment and related indices) depends on the users' contributions and their collective intelligence [20].

The central idea is to respect and exploit the fact that for some tasks humans can provide excellent help. The challenge then rests on our ability to harness and properly aggregate individual input to derive the community wisdom and exploit it to solve the problem at hand. One particularly interesting problem is that of constructing taxonomies. Taxonomies provide great help for structuring and categorizing our data sets. As such, currently they are at the heart of many web applications: Products are available that exploit taxonomic knowledge in order to improve results in product search applications [1, 2]. In local search, location taxonomies are used to improve results by utilizing a querying user's location and the known location of search result items to improve result quality. Google's search news, localized results, and product categories is a prime example for these. As another example, domain-specific (a.k.a. vertical) search engines utilize taxonomies (topic hierarchies) which are browsable and searchable using complex queries and receiving ranked results.

Our work rests on the realization that social media users (contributing and annotating content) have a good understanding of the fundamental (subtype-supertype) relationships needed to define a taxonomy for their content. For instance, biologists collectively can help define the taxonomy to be used, for example in a vertical (biological) search engine. Users from different locations can collectively define a locality taxonomy used in search engines with localization services. Traders in e-shops can easily collectively come up with product categorizations. These examples show that **humans can** offer great help! Further, the vast success enjoyed by a great number of social web applications, prove that **humans are willing** to provide such annotations. Hence, the human's willingness and ability can help solve a problem that is close to the heart of the semantic web community and which is addressed here: Crowdsourcing taxonomies inherently promise to provide a way to come up with high-quality taxonomies, based on the collective knowledge of its users, which will be dynamic and reflect the user-community's understanding of the data space.

2 Problem Statement, Rationale, and Challenges

Our model does not depend on any "experts". We adopt an automated approach, with the additional feature that users explicitly provide us with relations between the keywords (so-called "tags") they employ to annotate the content they share. Humans have a good understanding of the supertype-subtype relations between various thematic categories, since these naturally exist around them. So, we aim to exploit **extended tagging** and a categorization capability in order to develop high-quality taxonomies. We ask users to contribute with metadata in this format: $tag_a \rightarrow tag_d$. Here, tag_a is a supertype topic and represents a higher

node to a potential concept hierarchy whilst tag_d is a subtype topic. The arrow between them connotes an *Ancestor* $\rightarrow$ *Descendant* ($A \rightarrow D$) relation. In figure

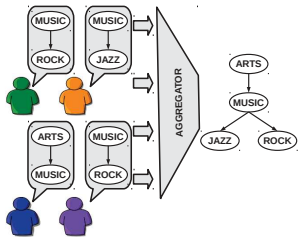


Fig. 1: Example

1 we demonstrate the basic idea of our effort. A community of users, forming a crowdsourcing environment, provides the system with tag relations. These can either be different between each other or depict the same relationship (i.e. Music $\rightarrow$ Rock). We refer to these tag relations also with the term **"votes"**, since they incorporate users' personal opinions for parts of the taxonomy tree. Our goal is to aggregate all given tag relations (votes) and produce a taxonomy that is derived using our community's knowledge/wisdom.

The problem is not easy! The following challenges arise:

- **Individuals are prone to errors.** Sometimes they specify incorrect tag relations, e.g., because of constrained knowledge and highly complex datasets. When building large scale taxonomies, the granularity level between nodes in adjacent levels is "fine" (especially at the lower levels of the taxonomy) and thus the frequency of such "false votes" may be high. So, contradicting opinions arise. But, this happens not only directly, but also indirectly, as a result of a combination of various relationships. Our goal is to resolve such conflicts. Since we depend on the crowd's wisdom, a natural discourse is to have the majority opinion prevail.
- **Incomplete (structural) information.** Users' knowledge might not be wide enough to completely cover an ideal, "golden rule" taxonomy (e.g. as constructed by experts). For example, in figure 1, suppose that a vote Arts $\rightarrow$ Jazz (which is valid) were given instead of the Arts $\rightarrow$ Music one. In this scenario we have evidence that both Arts and Music nodes are ancestors of node Jazz but we have no insight as far as their relative relationship is concerned. In this case, the system must be able to implicitly utilize the users' given input and fill in the missing structural information. This filling may be done incorrectly. Our approach will be to make a best guess (according to some metrics) and rely on future incoming votes to correct any such mistakes.
- **Incremental, online taxonomy development.** The problem is set in a dynamic environment. Users provide us with relations in an online way. The objective here is to introduce any newly incoming relations into the current structure in a cumulative manner. For every new vote, we shall be able to

modify and alter the current taxonomy state without having to destroy or build it from scratch. Based on the above example, suppose that we eventually receive an Arts → Music vote. If previously we had made a mistake when filling the lack of knowledge between Arts and Music, an efficient rectification must take place based on the now completed information. To sum up, dynamic, piece-wise, online taxonomy maintenance is is a key characteristic.

The "human-centric" approach we describe exploits users' knowledge (which is very difficult to correctly derive by automatic means) and produces taxonomies that are in accordance to the beliefs of the system's user base. In a sense, instead of constraining user input to characterizing the content (as is typically done in social media environments), we go one step further and allow users to provide with input that will lead to the construction of taxonomized datasets.

2.1 The Model, Solution Invariants, and Assumptions

As mentioned, the shape and structure of the taxonomy emerge from the crowd's *subjective* will, as it evolves. As the number of participants increases, in general, the higher the output quality becomes. When conflicts arise, several conflicting taxonomy states emerge as alternatives. One of them will be associated with the greatest number of votes. In this way, the new accepted state of the taxonomy will emerge. At the end, this process will converge to a structure, entirely defined from the community's aggregated knowledge. At this point, we can claim that this final product *objectively* depicts a complete taxonomy. But how do we evaluate such a taxonomy? We wish we could compare it against a golden rule taxonomy and see how they differ; but there is no standardized, "ideal" structure on which everyone agrees. Even if we compare taxonomies created by experts there will not be a 100% match, since both the rules for creating the taxonomy and the input data are often contradictory and obscure.

Users are asked to provide us with *Ancestor* → *Descendant* relations but any given vote has it's own interpretation, depending of the current state of the taxonomy. Any incoming tag relation will be classified to a category, based on the relative positions of the nodes it touches. For example, in figure 1 the following possible scenarios can arise as far as an incoming vote is concerned:

- **Ancestor → Descendant (A-D)**: i.e. Arts → Jazz. These relations practically increase our confidence for the current state of the taxonomy and generally leave it intact.
- **Descendant → Ancestor (D-A)**: i.e. Rock → Arts. These votes form what we call **backedges** and create cycles on the current structure. They require special handling and may change the current state of the taxonomy. They are not necessarily "false votes" according to a golden rule taxonomy and they are usefull, since they can restore possible invalid relations which were based on previous erroneous input or assumptions.
- **Crosslinks**: Relations like Jazz → Rock, do not belong to any of the former classes. This type of links interconnects nodes that have a common ancestor. If according to the golden rule taxonomy, there is a supertype-subtype

relationship between these nodes, our algorithms for handling this situation will be able to eventually yield the proper tree structure. If, however, there is no such relation between the nodes of a crosslink, then our algorithms will inescapably produce a supertype-subtype realtion between the two. When a relation like this arises, special handling is required: our current idea for this involves users supplying "negative" votes when they see incorrect crosslink relationships established in the current taxonomy. We leave discussion of this to future work.

Before we continue with the problem formulation we specify two **solution invariants** our approach maintains and give some insight of the taxonomy building algorithm that follows.

Tree Properties: This is the primary invariant we maintain. A tree is an undirected graph in which any two vertices are connected by exactly one path. There are no cycles and this is a principle we carry on throughout the taxonomy evolution. Starting with many shallow subtrees, as votes enter the system, we detect relations between more and more tags. The independent trees gradually form a forest and we use a common "Global Root" node to join them.

Maximum Votes Satisfiability: We also wish to preserve a *quantitative* characteristic. Our purpose is to utilize every incoming $Ancestor \rightarrow Descendant$ relation and embed it on the current structure. If this raises conflicts, our solution to this is to derive a taxonomy structure which as a whole satisfies the maximum number of users' votes. At this point we need to mention that according to our model, there is no constraint to the number of votes a user can submit to the system (see "free-for-all tagging" at [15]). Satisfiability is measured not on per individual basis but over all votes, overall.

Finally, we need to state that our solution does not take any measures to face synonyms or polysemy issues. Although according to [12] these are not major problems for social media, we admit that users tend to annotate their content with idiosyncratic tags which in our case can lead to wrong keyword interpretation and create links that users do not intent to recommend. This issue is orthogonal to our work since we focus on structural development and thus we can assume a controlled vocabulary without loss of generality.

3 Formal Formulation and Analysis

Leaving aside the added complexity due to the online nature of our venture, we show that even an offline approach yields an NP-Hard problem.

Note, that at first thought, one could suggest the following solution to our problem: First, create a directed graph $G(V, E)$ where each vertex $v \in V$ represents a given tag and each $e \in E$ represents a relationship between the two nodes. Edges bear weights w reflecting the number of votes from users for this relation. Intuitively, this calls to mind minimum/maximum spanning tree algorithms. Thus, second, run a "variation" of one of any well-known algorithms to retrieve a Maximum Spanning Tree. Because, however, our graph is directed, what we need here is a 'maximum weight arborescence' (which is defined as the

directed counterpart of a maximum spanning tree). However, this simplistic idea has a major flaw. If the graph is not strongly connected (something that regularly happens especially during the early stages of the taxonomy development) then a number of nodes has to be omitted from the final output.

Our problem is formalized as follows:
INPUT: Complete graph $G = (V, E)$, weight $w(e) \in Z_0^+$ for each $e \in E$.
OUTPUT: A spanning tree T for G such that, if $W(\{u, v\})$ denotes the sum of weights of the edges on the path joining u and v in T, then find B where:

$$B = max(\sum_{u,v \in V} W(\{u, v\})) \tag{1}$$

This problem is a straightforward instance of the Optimum Communication Spanning Tree problem in [9] and has been proven to be NP-Hard[1]. The key idea here for the matching of the two problems, is that **shortcuts** (edges weighting 0) are permitted. Obviously, as an algorithm proceeds online maintaining the tree invariant, there will be cases when nodes are connected with their edge having zero weights (as users may have not supplied yet any votes for this edge). B represents the maximum number of votes that is satisfied and along with the tree notion meets the standards set by the invariants in the previous section.

4 Crowdsourced Taxonomy Building Algorithm

As noted, our problem is NP-Hard. For n nodes, an optimal solution would require an exhaustive search of all possible n^{n-2} spanning trees and the selection of the one that maximizes value B in (1). So, we adopt a heuristic approach. We relax the second invariant: we demand the maximum number of satisfied votes, not overall, but only between consecutive algorithmic steps. Each vote is embodied into the taxonomy via an algorithm that introduces a series of transformations for every incoming vote (tag pair).

4.1 The Core Algorithm: CrowdTaxonomy

Algorithm 1 is called on every incoming vote. For every vote $(u \rightarrow v)$ we first need to identify whether the named nodes are new to the system or if they are already part of the tree. In case both nodes already exist (line: 9) we need to specify their relative position and thus we call a Lowest Common Ancestor (LCA) routine that returns the LCA node w. If $w = null$ (line: 11), there is no common ancestor and nodes u and v belong to different trees. If w coincides with u (line: 14), then u is already an ancestor of v, which is something that strengthens the evidence we have for the current state of the taxonomy. When w equals v (line: 16) the $v \rightarrow u$ relation introduces a conflict and implies that a

[1] We consider the requirements equal to the standard basis vector and refer to the Optimization version

modification may be needed. If we accept this edge, the structure's constraints are violated since a cycle is created.

Algorithm 1 VOTE PROCESSING

Require: A vote $tag_x \rightarrow tag_y$

```
 1: Node u ← hash(tag_x)
 2: Node v ← hash(tag_y)
 3: if ((u = null) and (v = null)) then
 4:    CREATE NEW TREE
 5: else if ((u ≠ null) and (v = null)) then
 6:    ATTACH NEW CHILD
 7: else if ((u = null) and (v ≠ null)) then
 8:    MERGE
 9: else
10:    Node w ← LCA(u, v)
11:    if (w = null) then
12:       MERGE
13:    else if (w = u) then
14:       CREATE FORWARD EDGE
15:    else if (w = v) then
16:       BACKEDGE CONFLICT RESOLUTION
17:    else
18:       EXPAND VERTICALLY
19:    end if
20: end if
```

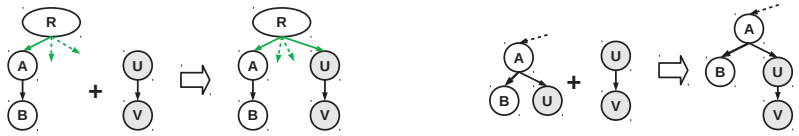
Lastly, in case w is a separate node on the tree (line: 18), the new relation forms a crosslink and is handled appropriately. Hereafter, we describe every tree transformation triggered by each of these cases.

TRSFM CREATE NEW TREE**:** In this simple scenario the taxonomy does not yet include any of the two nodes of the new vote. So the ancestor node u is attached to the global root via a *shortcut* ($R \rightarrow u$) and node v plays the role of its child (see figure 2a.).

Definition 1 ***Shortcut:*** *An "artificial" Parent → Child link that is not an explicit user supplied vote,. Its weight is 0, and it is utilized to preserve structural continuation.*

This addition forms a new tree with only two nodes. In the future, it will be expanded with more nodes or get merged to another expanding tree.

TRSFM ATTACH NEW CHILD**:** This is another straightforward case. Node u pre-exists and the incoming relation can be easily assimilated by adding an extra child to it.



(a) Create New Tree (b) Attach New Child

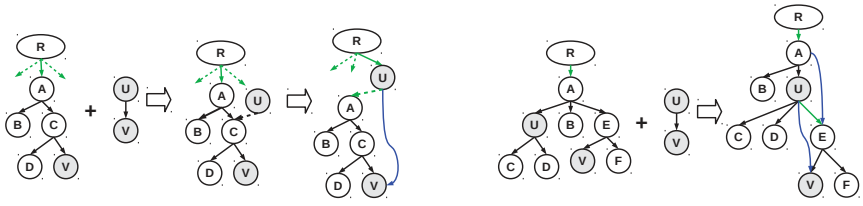
Fig. 2: Creating a new tree or attaching a new child

TRSFM Merge**:** Merge is used in two similar cases. In line 8 of the core algorithm we ask to attach a new node u to our taxonomy but in a generic scenario its descendant node v does already have a parent node. So does happen in line 12 where we need to annex u's participating tree to that one of v's with a link between them. In figure 3a we observe that both node C and u "compete for the paternity" of v. In order to maintain the tree properties, only one of the potential parents can be directly connected with v. We arbitrarily choose node C to be the direct ancestor (parent) of v and set node u to be parent of C which is in accordance with the *Maximum Vote Satisfiability* invariant since an *Ancestor* $\rightarrow$ *Descendant* ($u \rightarrow v$) relation takes place. The (temporary) state formed in the middle of figure 3a suffers from the same "paternity conflict" problem - now between A and u over C. Following the same reasoning, we finally place node u on top of v's tree being now the parent of v's root. Since there is no given relation yet between u and A we form a shortcut between them. We also maintain a *forward edge* from u to v so not to lose the information we have regarding the vote for the $u \rightarrow v$ relationship.

Definition 2 ***Forward Edge:*** *A latent relation between two nodes. The source node is an ancestor in the taxonomy and the target is a descendant. Forward edges do not refer to Parent* $\rightarrow$ *Child links and remain hidden since they violate tree's properties.*

The idea behind this transformation is that since we don't have enough evidence to decide on the partial order of v's ancestors we *temporarily* send u node to the root. Relations that will follow will illustrate the correct order.

If v is a root of a tree, an Attach New Child transformation is called.



(a) Merge (b) Expand Vertically

Fig. 3: The Merge and Expand Vertically transformations

TRSFM Expand Vertically**:** In this case the newly incoming vote is interpreted as a crosslink according to the current state of the taxonomy. As shown in figure 3b the logic we follow resembles at some point the Merge transformation. First, we locate the common ancestor A of u and v and break the link between it and the immediate root E of the subtree that includes v. The independent subtree is now linked with u via a shortcut formed between u and its root E. Two forward edges are spawned to indicate latent $Ancestor \rightarrow Descendant$ relations. For the sake of completeness we report here that in contrast to *common* edges between nodes, shortcuts are not converted to forward edges when the link that touches the two nodes brakes.

TRSFM Create Forward Edge**:** Straightforward scenario. A forward edge is added from u to v unless their node distance is 1 ($Parent \rightarrow Child$). In any other case (distance = 1) no operation takes place.

TRSFM Backedge Conflict Resolution **(BCR):** At line 16 of the Vote Processing algorithm we need to handle a vote whose interpretation is against the relationship of the nodes it touches in the current taxonomy state and any attempt to adjust it, leads to a backedge.

Definition 3 ***Backedge:*** *A latent relation between two nodes. The source node appears as a descendant in the current taxonomy whereas the target as an ancestor. Backedges remain hidden since they violate the tree's properties.*

Besides the cycle that it creates, a backedge might also violate the *Maximum Votes Satisfiability* invariant. The idea to solve this problem is to isolate the strongly connected component that the newly incoming backedge created and resolve any vote conflict locally.

We will present the logic of this transformation with a specific example on figure 4. For the sake of simplicity we assume that the incoming vote appears in a "triple form" and will be processed as such. On the left we observe the initial state of the isolated subgraph. We are asked to embody a $u \rightarrow v$ relation whose weight (number of votes) equals 3. We apply a what-if analysis. We instantiate all possible states, which the subraph component can reach, every time we apply a vertical rotation to its nodes. These are presented on the right part of the figure. The state whose backedges add up to the minimum weight is picked as a final state. In this example we arbitrary choose between state 3 or 4. Node A and its in- or out- going edges does not take part into this computation and is displayed here just to illustrate how it interfaces with the rest of the graph.

Formally, the aforementioned is an instance of the *All-pairs Bottleneck Paths* problem [19] and aims to find out the state in which we displease (or dissatisfy) the least number of votes. It's dual equivalent problem is the *All-pairs Maximum Capacity* and can be respectively interpreted as an effort to preserve our second invariant (*Maximum Votes Satisfiability*).

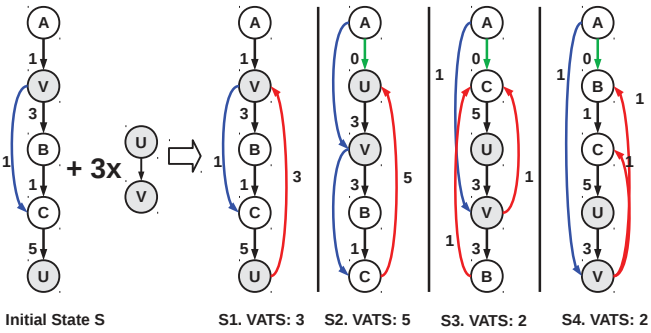


Fig. 4: Backedge Conflict Resolution What-if Analysis. VATS: Votes Against This State

Elimination of rising crosslinks

In fig. 5 we observe the big picture during a BCR. The change caused by the appearance of $(u \to v)$, practically decomposes the taxonomy into 3 components. The one on top, T, right above v, remains put. The lower ones are being inverted after we identify and break the weakest edge between them. This transformation has some corner cases. Former forward edges, such as $b \to d$ now appear as crosslinks, violating the first invariant. A solution to this anomaly is offered by algorithm 2, which simply certifies that all forward edges on path $\{v, ..., b\}$ adhere to the obvious ancestor-descendant relation.

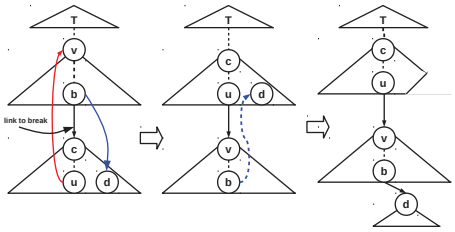


Fig. 5: The big picture of BCR and the algorithm for crosslinks elimination

Algorithm 2 Crosslink Elimination

1: **for all** Nodes $b_i \in$ **{v...b} do**
2: **for all** forward edges (b_i, d) **do**
3: **if** $(b_i \neq \text{LCA } (b_i, d))$ **then**
4: Expand Vertically(b_i, d)
5: **end if**
6: **end for**
7: **end for**

4.2 Asymptotic Complexity Analysis

The core Algorithm 1 is called once for every new vote. In turn, it calls at most one of the transformations. Therefore, its worst-case asymptotic complexity is equal to the worst of the worst-case complexities of each of the transformations. Having computed all these partial complexities we state that the worst one, is the one corresponding to Backedge Conflict Resolution. Every time a backedge appears it interconnects two nodes and the path connecting them

(coined BCR path) has a maximum length of n, where n denotes the number of total nodes. This is an extreme scenario where all the tree nodes form a chain. As we already stated, at this point, the BCR algorithm forms n possible tree instances, with every one of them representing a unique cyclic rotation of the nodes making up the BCR path. For every one of these n instances, we iterate over the n nodes it consists of and explore their outgoing edges to verify whether they form a backedge or not. The number of these outgoing edges is obviously also at most n-1. Therefore, the overall asymptotic worst-case complexity of the Backedge Conflict Resolution transformation is $O(n^3)$.

Theorem 1. *If s denotes the number of votes, the worst-case asymptotic complexity of CrowdTaxonomy Algorithm is $O(s * n^3)$.*

Proof. Vote Processing is called s times, once for each incoming vote. Every time a single transformation is applied. The worst-case complexity of the latter equals $O(n^3)$. Therefore the worst-case asymptotic complexity of the CrowdTaxonomy algorithm is $O(s * n^3)$.

This analysis depicts a worst-case scenario and is basically presented for the sake of completeness, vis a vis the NP-Hard result presented earlier. As the experiments showed, the algorithm's behaviour in matters of absolute time is approximately linear to the number of votes. This is because (i) BCR paths are much smaller than n and (ii) outgoing links per node are also much smaller than n. In future work we plan to present an analysis of the average complexity and provide with better estimations than n which is a very relaxed upper-bound.

5 Experimentation

We evaluate the CrowdTaxonomy algorithm and demonstrate its quality under: i) Lack of (structural) information and ii) the presence of conflicting votes. Further, we perform a real world crowdsourcing experiment: we test our initial assertion, that users are capable of providing valuable input when creating a taxonomy. We also test the quality of the taxonomy produced by CrowdTaxonomy with real-world input. Our metrics for evaluating the resulting taxonomy are based on the similarity between it and the golden rule one. Consider a taxonomy T_o as the golden rule taxonomy, and T as the one derived from our algorithm. Let R_o be the set of all $Parent \rightarrow Child$ relations on the T_o and R the set of all $Parent \rightarrow Child$ relations on T. We define:

$$Recall = \frac{|R_0 \cap R|}{|R_0|}, Precision = \frac{|R_0 \cap R|}{|R|}, FScore = \frac{2 \cdot Recall \cdot Precision}{Recall + Precision}$$

$|S|$ denotes the cardinality of a set S.
Recall certifies the completeness of the taxonomy, whereas *Precision* measures its correctness. *FScore* is a combination of the former into a harmonic mean.

We use the ACM Computing Classification System (version 1998) as the golden rule taxonomy. It contains 1473 concepts, forming a four-level tree. We

"break" this tree into distinct node (pair) relations, generate additional conflicting pairs and feed (correct and incorrect pairs) to our algorithm.

Our algorithm was written in C and we used GLib. The interface of the crowdsourcing experiment was implemented in HTML/PHP and ran over Apache/MySQL.

5.1 Synthetic Experiments

A key challenge we face is to provide a high-quality taxonomy under incomplete structural information. The votes are given ad-hocly and there is no guarantee that they form a complete taxonomy. To examine this characteristic we form all possible *Ancestor* → *Descendant* relations of the ACM tree and gradually feed the algorithm with a fraction of them. The result is shown in figure 6.

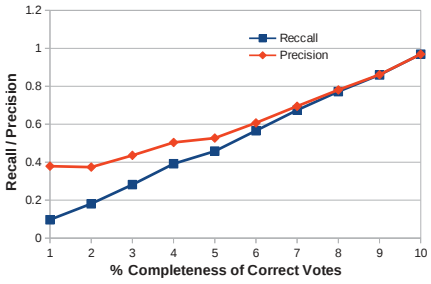


Fig. 6: Recall and Precision using a percentage of Correct votes

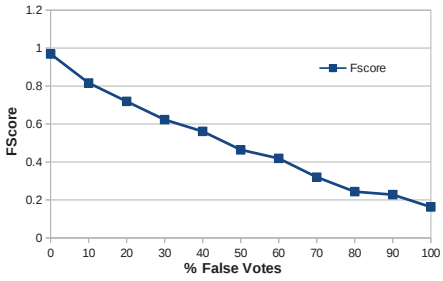


Fig. 7: FScores having a mixture of Correct and False votes

We repeatedly increase by 10% the number of available votes and measure the quality of our taxonomy. The absence of *Descendant* → *Ancestor* relations keeps the number of *permanent* backedges to zero. Any backedges that arise are instantly resolved and one, and only one, tree instance (representing 0 conflicts) is produced. The Recall value is linear to the size of input as expected. The Precision metric equals 0.4 at the begining and gradually increases. The taxonomy edges consist of a set of correct edges and a number of additional shortcuts, with a fraction of them playing a role of 'noise'. As we reach 100% input completeness, the assumptions (shortcuts) are gradually eliminated and both metrics end to 1.0 verifying a sanity-check, since the experiment configuration deals with only correct votes. Next we 'infect' the input with additional false votes. The number of correct *Ancestor* → *Descendant* votes remains 100% but on every iteration we increase the number of false (*Descendant* → *Ancestor*) votes. We form these by reversing the direction of a relation between two nodes. In figure 7 we observe that even with a high number of bad input, the taxonomy quality is high. For instance, even with 30% false votes, the F score remains above 60%. Under normal circumstances, where users can provide a largely complete and correct set of A-D relations, our algorithms produce a high quality taxonomy. Note that in this configuration $Recall = Precision = FScore$.

5.2 Crowdsourcing Experiment

We asked students of our department to voluntarily participate in crowdsourcing a taxonomy. We pregenerated all the possible *Ancestor* → *Descendant* relations and derived all the 'false' *Descendant* → *Ancestor* counterpart votes. We removed all misleading concepts (e.g., with labels 'General' or 'Miscellaneous') and attached their child nodes to their direct ancestors. As the total number of possible A-D relations is very high we used a fraction of the tree with 245 nodes which led to 620 relations (A-D) plus their counterparts (D-A). The users were presented with pairs of votes, the original correct vote, and the (inverted) false one. They had to choose between the one that depicted a $A \rightarrow D$ relation, linking two concepts of the ACM tree nodes. The user task consisted of 25 pairs of votes that were presented in groups of 5. Users could also select not to answer a vote-question marking it as *unspecified*. We counted 102 distinct http-sessions but the number of collected votes was smaller than the theoretical (2550), since some users dropped out early. In contrast to commercial crowdsourcing hubs, our user base had no financial or other type of gain. We collected input for a 3-day period. Some statistics of the experiment are shown on Table 1. We observe that the false/correct votes ratio is 0.283, which testifies that users are capable of providing high-quality input. Overall, the input accounted for 94.7% of all correct relations. With these statistics, we ran a synthetic experiment with controlled input, which predicted an $FScore$ ($Predicted$) whose value is close to the one in the real-world experiment. Thus, we can conclude that (i) the real-world experiment corroborated our conclusions based on the synthetic one; (ii) users can indeed provide high-quality input en route to a crowdsourced taxonomy, and (iii) despite the voluntary nature of user participation and the heuristic nature of our algorithms, the end result is a taxonomy preserving the large majority of the relationships found in expertly constructed taxonomies.

Total Votes	2155
Correct Votes	1501
False Votes	435
Unspecified	229
FScore	0.486
FScore (Predicted)	0.519

Table 1: Crowdsourcing Experiment Statistics

6 Related Work

[4], [5], [18] and [17] apply association mining rules to induce relations between terms and use them to form taxonomies. For text corpora, Sanderson and Croft automatically derive a hierarchy of concepts and develop a statistical model

where term x subsumes term y if $P(x|y) \geq 0.8$ and $P(y|x) < 1$ where $P(a|b)$ defines the probability of a document to include term a assuming that term b is contained. Schmitz extended this and applied additional thresholds in order to deal with problems caused by uncontrolled vocabulary. [6] and [14] underline the importance of folksonomies and the need to extract hierarchies for searching, categorization and navigation purposes. They present approaches that operate based on agglomerative clustering. A similarity measure is used to compute the proximity between all tags and then a bottom-up procedure takes place. Nodes under a threshold are merged into clusters in a recursive way and eventually compose a taxonomy. Heyman & Garcia-Molina in [11] present another technique with good results. Given a space of resources and tags, they form a vector for every tag and set to the $i-th$ element the number of times it has been assigned to object i. They also use cosine similarity to compute all vectors' proximities and represent them as nodes of a graph with weighted edges that correspond to their similarity distance. To extract a taxonomy they iterate over the nodes in descending centrality order and set every of its neighbours either as children or to the root based on a threshold.

As Plangprasopchok et al. note in [16] all these approaches make the assumption that frequent words represent general terms. This does not always hold and any threshold tuning approach leads to a trade-off between accurate but shallow taxonomies against large but noisy ones. Also, all above works assume a static tag space, despite its dynamicity [10].

7 Conclusions

We have presented a drastically new approach to create taxonomies, exploiting the wisdom of crowds and their proven desire and ability to provide rich semantic metadata on several social web applications. Our contributions include the definition and analysis of the problem of crowdsourcing taxonomies. We showed how to model the problem and the required human input and the (meaningful in a crowdsourcing environment) invariant of maximum vote satisfiability. Then we proceeded to show that the resulting problem is NP-Hard. Next, we contributed a novel heuristic algorithm to online aggregate human input and derive taxonomies. We conducted both synthetic and real-world crowdsourcing experiments. Our synthetic experiments showed that when the human input is adequately complete and correct, our solution can derive high-quality taxonomies. Conversely, even when the input is incomplete and incorrect to a significant extent, still a good quality taxonomy can be constructed. Our real-world crowdsourcing experiment additionally showed that indeed humans can provide high quality input in terms of completeness and correctness (at least for the taxonomy examined). And as our synthetic results showed, when fed into our algorithms, good quality taxonomies emerge. To the best of our knowledge this is the first work to study this problem and provide a promising solution.

Future work includes optimizations, straddling the complexity-quality trade-offs, and appropriate measures for crowdsourced taxonomy quality evaluation.

References

1. Endeca. `http://www.endeca.com/`.
2. Facetmap. `http://http://facetmap.com/`.
3. O. Alonso and M. Lease. Crowdsourcing 101: Putting the wsdm of crowds to work for you: A tutorial. In *International Conference on WSDM, February*, 2011.
4. C.-m. Au Yeung, N. Gibbins, and N. Shadbolt. User-induced links in collaborative tagging systems. In *Proceeding of the 18th ACM conference on Information and knowledge management*, CIKM '09.
5. M. Barla and M. Bieliková. On deriving tagsonomies: Keyword relations coming from crowd. *Computational Collective Intelligence. Semantic Web, Social Networks and Multiagent Systems*, 2009.
6. C. H. Brooks and N. Montanez. Improved annotation of the blogosphere via autotagging and hierarchical clustering. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06.
7. A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the world-wide web. *Commun. ACM*, 2011.
8. M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In *ACM SIGMOD Conference*, 2011.
9. M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*.
10. H. Halpin, V. Robu, and H. Shepherd. The complex dynamics of collaborative tagging. In *16th WWW Conference*, 2007.
11. P. Heymann and H. Garcia-Molina. Collaborative creation of communal hierarchical taxonomies in social tagging systems. Technical report, 2006.
12. P. Heymann, A. Paepcke, and H. Garcia-Molina. Tagging human knowledge. In *Third ACM International Conference on Web Search and Data Mining (WSDM2010)*.
13. P. Ipeirotis. Managing crowdsourced human computation: A tutorial. In *International Conference on WWW, March*, 2011.
14. K. Liu, B. Fang, and W. Zhang. Ontology emergence from folksonomies. In *19th ACM CIKM*, 2010.
15. C. Marlow, M. Naaman, D. Boyd, and M. Davis. Position Paper, Tagging, Taxonomy, Flickr, Article, ToRead. In *Collaborative Web Tagging Workshop at WWW2006*.
16. A. Plangprasopchok, K. Lerman, and L. Getoor. Growing a tree in the forest: constructing folksonomies by integrating structured metadata. In *6th ACM SIGKDD Conference*, 2010.
17. M. Sanderson and B. Croft. Deriving concept hierarchies from text. In *22nd ACM SIGIR Conference*.
18. P. Schmitz. In *WWW 2006*.
19. A. Shapira, R. Yuster, and U. Zwick. All-pairs bottleneck paths in vertex weighted graphs. In *18th ACM-SIAM SODA Symposium*, 2007.
20. P. Triantafillou. Anthropocentric data systems. In *37th VLDB Conference, (Visions and Challenges)*., 2011.