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-

(c) 2014, Copyright is with the authors. Published in Proc. 17th International Conference on Extending Database Technology (EDBT), March 24-28, 2014, Athens, Greece: ISBN 978-3-89318065-3, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0
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.

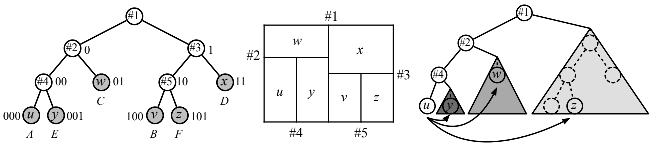
Wang et al. present in [18] SSP (Skyline Space Partitioning) for distributed processing of skyline queries in BATON [10]. The multi-dimensional data space is mapped to unidimensional keys using a Z-curve, due to BATON limitations. Query processing starts only at the peer responsible for the region containing 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.link$[i]$.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: $S_w^L \leftarrow w$.computeLocalState$(Q, S^G)$
2: $S_w^G \leftarrow w$.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$\cap R)$
6:     end if
7: end for
8: $A \leftarrow w$.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:  $S^L_w \leftarrow w$.computeLocalState($Q, S^G$)
2:  $S^G_w \leftarrow w$.computeGlobalState($Q, S^G, S^L_w$)
3:  for $\ell \in w$.sortLinks($w$.comp($i, j, Q$)) do
4:      if $w$.isLinkRelevant($\ell, Q, S^G_w, R$) then
5:          $w$.link[$\ell$].slow($v, w, Q, S^G_w, w$.link[$\ell$].region$\cap R$)
6:          $S^L \leftarrow w$.receiveRemoteLocalState()
7:          $S^L_w \leftarrow w$.updateLocalState($Q, \{S^L_w, S^L\}$)
8:          $S^G_w \leftarrow w$.computeGlobalState($Q, S^G, S^L_w$)
9:      end if
10: end for
11: $w$.sendLocalStateTo($u, S^L_w$)
12: $A \leftarrow w$.computeLocalAnswer($Q, S^L_w$)
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.

```
$S_w^L \leftarrow w$.computeLocalState($Q, S^G$)
$S_w^G \leftarrow w$.computeGlobalState($Q, S^G, S_w^L$)
if $r > 0$ then
    for $\ell \in w$.sortLinks($w$.comp($i, j, Q$)) do
        if $w$.isLinkRelevant($\ell, Q, S_w^G$) then
            $w$.link[$\ell$].ripple($v, w, Q, S_w^G, w$.link[$\ell$].region$\cap R, r - 1$)
            $\{S_i^L\} \leftarrow w$.receiveRemoteLocalState()
            $S_w^L \leftarrow w$.updateLocalState($Q, \{S_w^L, \{S_i^L\}\}$)
            $S_w^G \leftarrow w$.computeGlobalState($Q, S^G, S_w^L$)
        end if
    end for
else
    for each link $i$ do
        if $w$.isLinkRelevant($i, Q$) then
            $w$.link[$i$].ripple($v, u, Q, S_w^G, w$.link[$\ell$].region$\cap R, 0$)
        end if
    end for
end if
$w$.sendLocalStateTo($u, S_w^L$)
$A \leftarrow w$.computeLocalAnswer($Q, S_w^L$)
$w$.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 $\tau^G$
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 $\tau_w^L$
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 $\tau_i^L$ values
2: $m_w^L \leftarrow 0$
3: for each entry $(m_i^L, \tau_i^L)$ 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 $(m_w^L, \tau_w^L)$
```

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)

```
return $A \leftarrow$ local tuples of $S_w^L$
```

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

```
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.

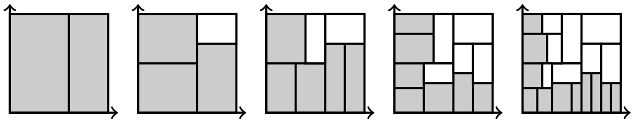

(a) level 1 (b) level 2 (c) level 3 (d) level 4 (e) level 5

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 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.

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

```
for each s $\in S_w^G$ do
    if s $\succ w$.link$[i].region$ then
        return false
    end if
end for
return true
```

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

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

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.

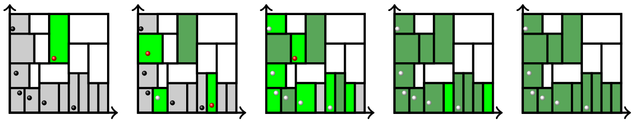

(a) start (b) 1st hop (c) 2nd hop (d) 3rd hop (e) 4th hop

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.,

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

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: $\mathbf{t} \leftarrow w$.getMostDiverseLocalObject($\mathbf{q}, O, \phi$)
2: if $\phi(\mathbf{t}, \mathbf{q}, O) < \tau^G$ then
3:     return $\tau_w^L \leftarrow \phi(\mathbf{t}, \mathbf{q}, O)$
4: else
5:     return $\tau_w^L \leftarrow \tau^G$
6: end if
```

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

```
1: return $\tau_w^G \leftarrow \tau_w^L$
```

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: $\mathbf{t} \leftarrow w$.getMostDiverseLocalObject($\mathbf{q}, O, \phi$)
2: if $\phi(\mathbf{t}, \mathbf{q}, O) = \tau_w^L$ then
3:     return $A \leftarrow \mathbf{t}$
4: else
5:     return $A \leftarrow null$
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 $\tau_w^L \leftarrow \min_i\{\tau_i^L\}$
```

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

```
1: return $\phi^-(w$.link[$i$].region, $\mathbf{q}, O) < \tau_w^G$
```

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

```
1: return $\phi^-(w$.link[$i$].region, $\mathbf{q}, O) < \phi^-(w$.link[$j$].region, $\mathbf{q}, O)$
```

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 $\phi(\mathbf{t}_i, q, O \smallsetminus \{\mathbf{t}_i\})$, 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 $O \smallsetminus \{\mathbf{t}_i\}$, 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., $\phi(\mathbf{t}_i, \mathbf{q}, O \smallsetminus \{\mathbf{t_i}\}) \geq$

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

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

```
O ← $v$.initialize($\mathbf{q}, k$)
for $i \leftarrow 1$ to MAX_ITERS do
    $O' \leftarrow v$.div-improve($\mathbf{q}, O$)
    if $O' \neq O$ then
        $O \leftarrow O'$
    else
        break
    end if
end for
```

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

```
1: $\mathbf{t}_{in} \leftarrow null$
2: $\mathbf{t}_{out} \leftarrow null$
3: sort tuples in $O$ descending on their $\phi$ scores
4: for each $\mathbf{t}_i \in O$ do
5:     if $\mathbf{t}_{in} = null$ then
6:         $\tau \leftarrow \phi(\mathbf{t}_i, \mathbf{q}, O)$
7:     else
8:         $\tau \leftarrow f(O \smallsetminus \{\mathbf{t}_{out}\} \cup \{\mathbf{t}_{in}\}, \mathbf{q}) - f(O, \mathbf{q})$
9:     end if
10:    $v$.div-ripple($v, v, \mathbf{q}, O \smallsetminus \mathbf{t}_i, \tau, R, r$)
11:    $\mathbf{t}_{in} \leftarrow v$.receive()
12:    if $\mathbf{t}_{in} \neq null$ then
13:        $\mathbf{t}_{out} \leftarrow \mathbf{t}_i$
14:    end if
15: end for
16: return $O \smallsetminus \{\mathbf{t}_{out}\} \cup \{\mathbf{t}_{in}\}$
```

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

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

Therefore, $O \smallsetminus \{\mathbf{t_i}\}$ 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 $O \smallsetminus \{\mathbf{t}_i\}$ 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.

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

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.

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.

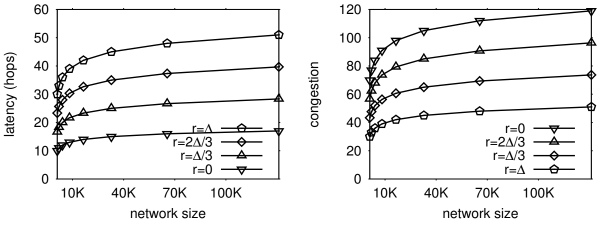
Dimensionality affects performance only slightly, as shown in Figure 5. The reason is that the core structure (number of neighbors per peer, overlay size) of the underlying index (MIDAS) determining performance is not affected; only the dimensionality of the zones changes. Finally, Figure 6 shows that 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.

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.

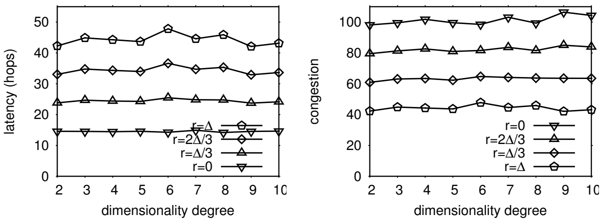
The evaluation of RIPPLE on skyline queries is shown in Figures 7 and 8 using the NBA and SYNTH datasets, respectively. In Figure 7(a) latency shows a logarithmic behavior for ripple-slow and SSP, due to the exploited properties of their indexing infrastructures. Nevertheless, SSP is not as efficient 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



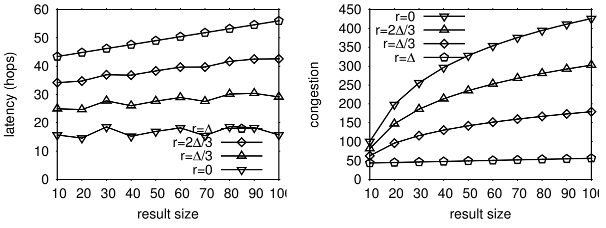
(a) latency (b) congestion

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.

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

[†] Available at http://press.liacs.nl/mirflickr/

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.

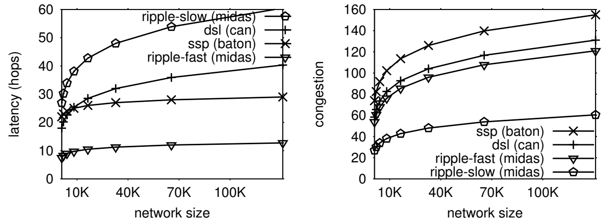
7.2.3 k-Diversification Queries

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.

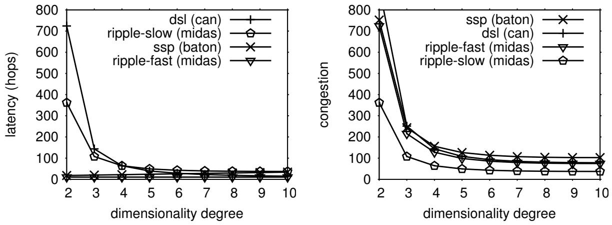
Moreover, the required number of iterations for the RIPPLE-based diversification algorithm to converge plays a prevalent role in the performance of the methods. Nevertheless, we note that performance is affected by both the effectiveness of the diversified search methods and the indexing infrastructure 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-



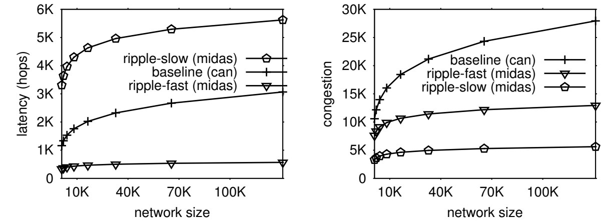
(a) latency (b) congestion

Figure 7: Skyline computation in terms of overlay size.



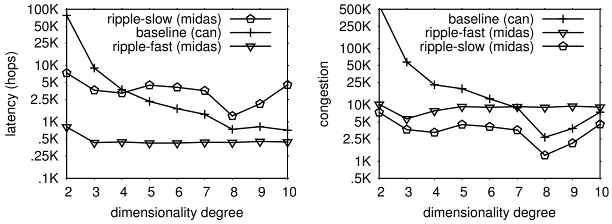
(a) latency (b) congestion

Figure 8: Skyline computation in terms of dimensionality.



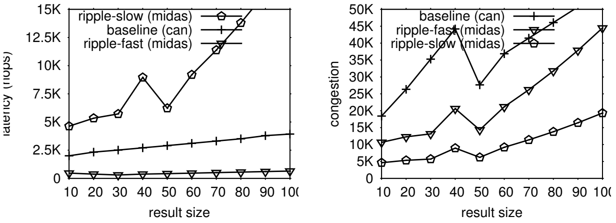
(a) latency (b) congestion

Figure 9: Diversification performance in terms of overlay size.



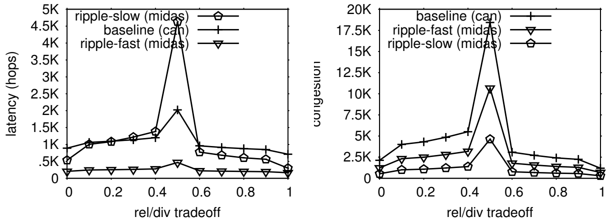
(a) latency (b) congestion

Figure 10: Diversification performance in terms of dimensions.



(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.