Continuous Monitoring of Top-k Queries over Sliding Windows

Kyriakos Mouratidis[1] Spiridon Bakiras[2] Dimitris Papadias[2]

[1]School of Information Systems
Singapore Management University
80 Stamford Road, Singapore 178902
kyriakos@smu.edu.sg

[2]Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{sbakiras, dimitris}@cs.ust.hk

ABSTRACT

Given a dataset P and a preference function f, a top-k query retrieves the k tuples in P with the highest scores according to f. Even though the problem is well-studied in conventional databases, the existing methods are inapplicable to highly dynamic environments involving numerous long-running queries. This paper studies continuous monitoring of top-k queries over a fixed-size window W of the most recent data. The window size can be expressed either in terms of the number of active tuples or time units. We propose a general methodology for top-k monitoring that restricts processing to the sub-domains of the workspace that influence the result of some query. To cope with high stream rates and provide fast answers in an on-line fashion, the data in W reside in main memory. The valid records are indexed by a grid structure, which also maintains book-keeping information. We present two processing techniques: the first one computes the new answer of a query whenever some of the current top-k points expire; the second one partially precomputes the future changes in the result, achieving better running time at the expense of slightly higher space requirements. We analyze the performance of both algorithms and evaluate their efficiency through extensive experiments. Finally, we extend the proposed framework to other query types and a different data stream model.

1. INTRODUCTION

Assume a dataset P, where each record p has d numerical attributes $p.x_1, \ldots, p.x_d$. A top-k query q specifies a preference function f that maps each tuple $p \in P$ to a real number $score(p) = f(p.x_1, \ldots, p.x_d)$. The result of q is the set of the k records with the highest scores. The top-k operator is important for several on-line applications, including communication and sensor networks, stock market trading, profile-based marketing, etc. Consider, for instance, an Internet Service Provider that monitors the traffic at various points (i.e., routers) inside the network. Monitoring tools, such as Cisco's NetFlow, generate and communicate (to a central server) detailed traffic logs on a per flow granularity (a flow consists of a series of packets having the same source and destination IP addresses). The resulting data streams have very high data rate, and usually account for hundreds of GBytes of data per day. A typical tuple in the above scenario includes the source and destination IP address, the start and finish time, the byte and packet count, etc. The availability of such records at the central server enables the continuous evaluation of numerous queries regarding traffic estimation, network security or troubleshooting. For example, one might want to monitor in real-time the top-100 flows with the largest individual throughput. If a number of results (i.e., flows) share the same destination IP address, this could be an indication that the destination node is the victim of an ongoing Distributed Denial of Service (DDoS) attack. On the other hand, one might ask "what are the top-100 flows with the minimum number of transmitted packets". Here, if a number of results share the same source IP address, it could be a sign of an Internet worm trying to spread itself (i.e., a source that randomly probes the IP address space – by sending TCP SYN packets – in order to discover vulnerable hosts). In both cases, the underlying application (i.e., network security) is time-critical and, thus, the timely and continuous evaluation of each query is essential.

As discussed in Section 2, various techniques have been proposed for *snapshot* top-k queries in conventional databases and distributed repositories. However, to the best of our knowledge, there is no algorithm for processing multiple long-running queries that request continuous evaluation in an on-line fashion. This paper addresses top-k monitoring over *sliding windows*, assuming the append-only data stream model [1]. In this context, tuples continuously stream in the system and they are considered *valid* only while they belong to a sliding window W. We consider two versions of windows: a *count-based* W contains the N most recent records, whereas a *time-based* W contains all tuples that arrived within a fixed time period covering the most recent timestamps. The task of the query processor is to constantly report the top-k set of every monitoring query among the valid data. We propose a general framework that applies to both count-based and time-based windows, to arbitrary k and dimensionality, and to any *monotone* scoring function.

In order to achieve real-time query evaluation, the valid tuples are stored in main memory and indexed by a regular grid. When a query q first arrives at the system, its

SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.

result is computed by the top-k computation module, which searches the minimum number of cells that may contain result records. Top-k maintenance deals only with tuple insertions and deletions that fall within these cells. We distinguish two policies and accordingly propose two algorithms: (i) the Top-k Monitoring Algorithm (TMA) re-computes the answer of a query q whenever some of the current top-k tuples expire; (ii) the Skyband Monitoring Algorithm (SMA) partially pre-computes future results, by reducing the problem to a *skyband* maintenance over a subset of the valid records. As we show analytically and verify experimentally, SMA has lower running time than TMA, at the expense of slightly higher space consumption.

The rest of the paper is organized as follows. Section 2 reviews previous work on top-k processing, as well as related query types. Section 3 provides some geometric observations that motivate our solutions, and describes a benchmark approach that combines existing work to solve the same problem. Sections 4 and 5 present the TMA and SMA methods, respectively. Section 6 analyzes the time and space complexity of the proposed algorithms. Section 7 discusses the application of our techniques to other preference-based queries, and extends our approach to update streams. Section 8 experimentally evaluates TMA and SMA. Finally, Section 9 concludes the paper with directions for future work.

2. RELATED WORK

Section 2.1 discusses existing top-k processing methods in various scenarios. Section 2.2 reviews continuous monitoring methods for related query types.

2.1 Top-k Query Processing

Onion [8] and *Prefer* [14] are preprocessing-based techniques for top-k queries in conventional databases. *Onion* is motivated by the fact that the point with the highest score (for any *linear* preference function) can be found within the convex hull of the dataset. The method computes and stores convex hulls in layers, with outer layers geometrically enclosing the inner ones. A linear top-k query is evaluated by processing the layers inwards, starting from the outmost hull. *Prefer* pre-computes a set of sorted lists according to some arbitrary scoring functions, and materializes them as views. For any given preference function f, it selects the materialized view corresponding to the function that is most similar to f; the query can then be answered efficiently by examining a subset of the records in this view. *Prefer* works for non-linear scoring functions, provided that a different set of views is maintained for each function type. Similar to *Onion*, *Prefer* is aimed mostly at static data (due to the high cost of pre-processing). Yi et al. [30] propose algorithms that reduce the storage and maintenance cost of materialized top-k views in the presence of deletions and updates. The basic idea of their methods is simple: instead of a materialized top-k view, a larger view containing $k' > k$ tuples is maintained. The value of k' varies between k and a system parameter k_{max} (that depends on the observed workload). During the initialization phase, a top-k_{max} query is processed and the view is filled with the retrieved entries. Incoming tuples whose score is larger than the score of the k'th tuple are inserted into the view. On the other hand, deleted tuples may reduce the number of entries k' below k. In this case, a top-k_{max} query is issued again on the database to refill the view with k_{max} entries.

Bruno et al. [5] utilize multidimensional histograms in relational databases to map top-k queries into traditional ranges. Similarly, Chen and Ling [10] use a sampling-based technique to transform top-k queries into approximate ranges. In both cases, if the range does not produce k results, the process is repeated. Another well-studied problem is reporting the top-k records among the results of a join operation over multiple relations. Donjerkovic and Ramakrishnan [11] apply probabilistic optimization to answer ranked queries involving selections and joins. Ilyas et al. [15] propose a pipelined algorithm, suitable for implementation inside a hierarchy of join operators. The same authors [16] also introduce a pipelined rank-join operator, based on the ripple join technique. Ilyas et al. [17] further explore the integration of rank-join operators in conventional database systems, by estimating their cost as part of a query execution plan. Finally, Tsaparas et al. [28] build a ranked join index to efficiently answer top-k join queries for arbitrary scoring functions.

Several papers focus on computing the top-k results from multiple (distributed) data repositories. As an example, consider that a user wants to find the k images that are most similar to a query image, where similarity is defined according to d features, e.g., color histogram, object arrangement, texture, shape, etc. The query is submitted to d retrieval engines and each engine returns a list of objects sorted in descending order of their partial scores (with respect to the corresponding feature). The problem is to compute the top-k results in terms of their overall similarity by combining the d sorted lists. The existing algorithms differentiate *sorted* and *random* accesses. Sorted access only supports the retrieval of the objects in descending order (of the partial score), while random access returns the score for any random element in the list. Fagin et al. [12] introduce two methods for processing ranked queries. The TA algorithm is optimal for repositories that support random access (it minimizes the number of random accesses). On the other hand, the NRA algorithm assumes that only sorted access is available. Variations of the methods have been proposed for several applications, including similarity search in multimedia repositories [9], approximate top-k retrieval with probabilistic guarantees [27], and ranked queries over web-accessible databases [21]. Finally, Chang and Hwang [7] introduce *MPro*, a general algorithm that optimizes the execution of expensive predicates for a variety of top-k queries.

The above methods assume that all the relevant data are available (locally, or in distributed servers) before processing. Further, they report a single result and terminate. On the other hand, in stream environments the data are not known in advance, but they keep changing as new tuples arrive and old ones expire. The objective is to continuously monitor the top-k tuples of long-running queries according to the record arrivals and expirations. The only relevant work in the data stream literature is by Babcock and Olston [2], who introduce the concept of *distributed top-k monitoring*. Their goal is to continuously report the k objects with the largest cumulative score over a set of stream sources. In order to reduce the communication cost, they maintain arithmetic constraints at the stream sources to ensure that the most recently reported answer remains valid. When a constraint is violated, the corresponding source reports it to the central server, which updates the top-k set and assigns new constraints to the sources. Our target problem is

different from [2] since we deal with multiple ordinary top-k queries over a single multidimensional stream. Furthermore, while Babcock and Olston aim at minimizing the network overhead, we aim at minimizing the CPU cost at the server side.

2.2 Continuous Monitoring for Related Query Types

Top-k queries are related to *skylines* and other similar problems such as *multi-objective optimization* and *maximal vectors*. A record p_1 is said to *dominate* another p_2, if and only if, p_1 is preferable to p_2 on every attribute. Informally, this implies that p_1 has a larger score than p_2 according to any preference function, which is monotone on all attributes. The skyline operator returns all tuples that are not dominated by another record. Skyline computation has received considerable attention in relational databases [4, 24] and web information systems [3]. Lin et al. [20] and Tao and Papadias [26] propose methods for skyline monitoring over sliding windows. The skyline maintenance is performed by in-memory algorithms, which discard records that cannot participate in the skyline until their expiration.

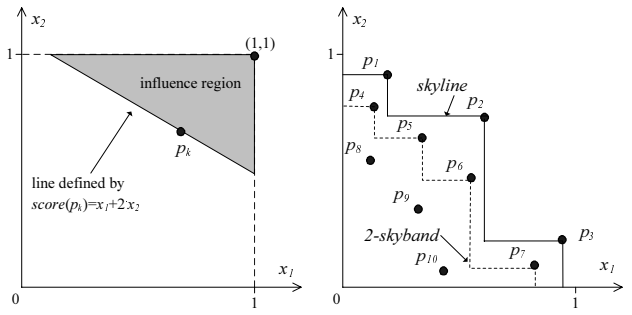
Top-k (and skyline) queries have a multidimensional aspect, since each record p can be thought of as a point[1] (in the d-dimensional space) defined by the attribute values $p.x_1, p.x_2, \ldots, p.x_d$. Therefore, top-$k$ monitoring can benefit from previous work on continuous spatial queries. The first monitoring method for spatial queries, *Q-index* [25], considers static range queries over moving objects. The queries are indexed by an R-tree and moving objects are probed against it in order to detect result changes. Kalashnikov et al. [18] show that a grid implementation of *Q-index* is more efficient (than R-trees) for main memory evaluation. Mokbel et al. [22] propose *Sina*, a system that centrally monitors moving range queries. Query evaluation in *Sina* is implemented as an incremental spatial join between the objects and the queries. *Mqm* [6] and *Mobieyes* [13] exploit the object computational capabilities in order to reduce the processing load of the central server. Koudas et al. [19] describe *Disc*, a technique for continuous evaluation of e-approximate k-NN (nearest neighbor) queries over streams of multidimensional points. Yu et al. [31] propose two grid-based methods for continuous monitoring of exact k-NN queries over moving objects. The first one indexes the data objects, and the second one indexes the queries. *Sea-Cnn* [29] and *Cpm* [23] maintain the result of continuous NN queries, by considering only object updates that may influence some query. Both methods use a grid and book-keeping information to determine the queries influenced by each cell.

3. PRELIMINARIES

The proposed methods apply to any top-k monitoring query, provided that its scoring function f is *monotone* on all attributes. A function f is increasingly monotone on dimension x_i if for any pair of tuples (points) p_1, p_2 with $p_1.x_i \geq p_2.x_i$ and $p_1.x_j = p_2.x_j \; \forall j \neq i$, it holds that $score(p_1) = f(p_1.x_1, p_1.x_2, \ldots, p_1.x_d) \geq score(p_2) = f(p_2.x_1, p_2.x_2, \ldots, p_2.x_d)$. Similarly, f is decreasingly monotone on x_i if for any pair of points p_1, p_2 as above it holds that $score(p_1) \leq score(p_2)$. Note that a function may be increasingly monotone on some dimensions, and decreasingly monotone on the remaining ones. Without loss of generality, we focus on 2-dimensional unit spaces (i.e., records with two attributes whose values range between 0 and 1) and scoring functions increasingly monotone on both dimensions. Section 3.1 describes some properties that permit the efficient computation and maintenance of the results, while Section 3.2 presents a competitor to our methods that combines previous work in order to process continuous top-k queries.

3.1 Properties

Consider the example of Figure 1(a), where p_k is the object with the kth highest score for a query q with function $f(x_1, x_2) = x_1 + 2 \cdot x_2$. The line defined by $score(p_k) = x_1 + 2 \cdot x_2$ divides the data space into two parts. The shaded area corresponds to the *influence region* of q; any update falling in the influence region will change the result of the query. In particular, an insertion will cause the removal of p_k and, consequently, the shrinking of the influence region, whereas a deletion (of a result record) will cause its expansion. On the other hand, updates outside the influence region are irrelevant to q since the corresponding tuples have a score below $score(p_k)$ and do not alter the result.



(a) Influence region (b) Skyline and skyband

Figure 1: Geometric observations on the top-k problem

For any scoring function f (increasingly monotone on x_1 and x_2), the point with the maximum score is at the corner (1,1) of the data space. In accordance with the skyline literature, point (1,1) dominates every other tuple. Similarly, all records falling in a rectangle R are dominated by its top-right corner. The score of this corner is denoted as $maxscore(R)$ and is an upper bound for the scores of all tuples in R.

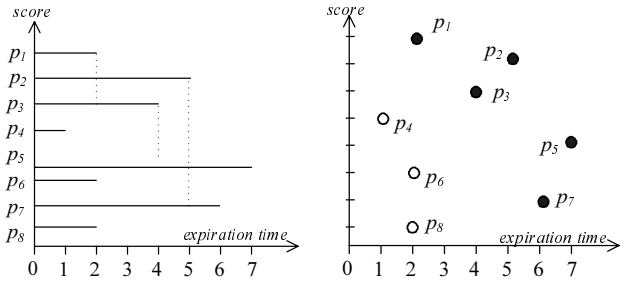
In general, the skyline of a dataset contains all tuples that belong to the result of any top-1 query with a monotone function. For instance, in Figure 1(b), the skyline includes p_1, p_2 and p_3, meaning that every top-1 query (on $p_1, \ldots, p_{10}$) returns one of these three tuples[2]. In order to generalize to arbitrary values of k, we use the concept of *k-skyband* [24]. Specifically, the k-skyband contains the tuples that are dominated by at most $k-1$ other points. According to this definition, the skyline is a special instance of the skyband, where $k = 1$. In Figure 1(b), the 2-skyband consists of all points $(p_1, \ldots, p_7)$ on, or to the right of, the

[1]Henceforth, the terms record, tuple and point are used interchangeably. Similarly, the data attributes are also referred to as coordinates.

[2]If queries are restricted to linear functions, the top-1 result belongs to the convex hull, which is a subset of the skyline. This fact is exploited by *Onion*, as discussed in Section 2.1.

dashed line. Tuples $(p_8, \ldots, p_{10})$ that do not belong to the 2-skyband cannot be in the result of any top-2 query, since they are always dominated by two or more points.

Now assume that each tuple is associated with an expiration time. Figure 2(a) illustrates an example, where at time $t = 0$ there exist 8 valid tuples. The horizontal axis corresponds to the lifespan of the records and the vertical one to their score according to a certain preference function. Assuming that there are no further arrivals, we can predict all future results. The top-2 set at time 0 is $\{p_1, p_2\}$. When p_1 expires at time 2, it is replaced by p_3. At time 4, p_3 expires and the result becomes $\{p_2, p_5\}$. Finally, at time 5, p_7 replaces p_2. The important observation is that *the records that appear in some result are the ones that belong to the 2-skyband in the score-time space*. Such records are shown solid in Figure 2(b). Consider, for instance, a tuple p that belongs to some (future) top-2 result. Then, there exists some time instance when p has a smaller score than (is dominated by) at most 1 other valid record. Therefore, p is part of the 2-skyband. Conversely, consider that p belongs to the 2-skyband in the *score-time* space. This implies that there is at most one record with score higher than p that expires after p. Thus, there exists some time instance when p is part of the top-2 result.



(a) Record lifespans (b) 2-skyband

Figure 2: Top-2 query as 2-skyband in *score-time* space

The reduction from top-k to k-skyband queries applies to both kinds of sliding windows (i.e., count-based and time-based ones) and is independent of the dimensionality d; i.e., the skyband is always computed in the 2-dimensional *score-time* space even if the records have numerous (> 2) attributes. This reduction is exploited by the SMA algorithm in Section 5. Next, we discuss an alternative method for continuous monitoring of top-k queries, which is built upon existing work. Due to the absence of other competitors, this method is used as benchmark for the proposed techniques.

3.2 Threshold Sorted List Algorithm

A top-k monitoring technique must integrate two modules: (i) a method for the initial computation of the top-k set, and (ii) an efficient maintenance mechanism to update the result in the presence of insertions/deletions. The Threshold Algorithm (TA) [12] (discussed in Section 2.1) is a good candidate for the first module, due to its popularity and good performance. For the second module, we use the technique of Yi et al. [30], which is applicable to any top-k computation method. We call this combined approach, the Threshold Sorted List (TSL) algorithm. Figure 3 shows the architecture of TSL, assuming a 2-dimensional space. All the valid tuples are stored in a first-in-first-out list where each entry p is a tuple $<p.id, p.x_1, p.x_2, p.t>$ (i.e., a unique id, the values of the two attributes, and the arrival time). In addition, TSL maintains 2 (in the general case d) lists of all points sorted on the two dimensions.

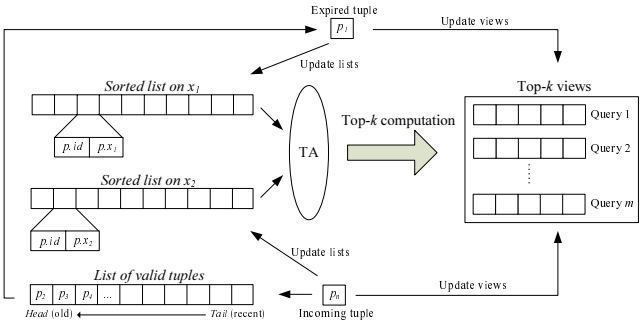


Figure 3: Threshold sorted list mechanism

When a new top-k query q arrives at the system, the TA module is invoked to compute the initial result, which contains $k_{max} > k$ entries (recall that [30] maintains more results than necessary in order to reduce re-computations). TA accesses the two sorted lists in a round robin fashion. For every retrieved point p, a random access is performed on the other attributes of p, and its score is computed. If p's score is larger than the current k_{max}th best score, p is added to the top-k_{max} result. Furthermore, after each round, a threshold τ is calculated based on the last attribute values encountered across all the sorted lists (i.e., τ is the maximum score that can be achieved by any non yet visited point). If the k_{max}th best score is larger than τ, the algorithm terminates and the view containing the k_{max} results is materialized (see Figure 3).

Upon arrival of a new data tuple p_n, the maintenance module of TSL (i) inserts $p_n.x_1$ and $p_n.x_2$ to the corresponding sorted lists, and (ii) computes its score for each of the m active queries. If any of the current m views is affected, its top-k' set is updated accordingly (k' is the number of entries in the current view and, at any time, $k \leq k' \leq k_{max}$). Specifically, p_n is inserted into the view and, if $k' = k_{max}$, the previous element at position k_{max} is removed. On the other hand, when an existing tuple (e.g., p_1) expires, TSL (i) deletes the corresponding entries from the two sorted lists, and (ii) removes p_1 from any of the m views that include it. If the number of entries in a certain view drops below k, the TA algorithm is invoked again to refill the view with k_{max} entries.

4. TOP-K MONITORING ALGORITHM

This section focuses on the Top-k Monitoring Algorithm (TMA). Section 4.1 describes the index and book-keeping structures, while Sections 4.2 and 4.3 present the top-k computation and maintenance modules of TMA, respectively.

4.1 Index and Book-keeping Structures

Assuming a 2-dimensional space, each record p is represented as $<p.id, p.x_1, p.x_2, p.t>$, where $p.id$ is a unique identifier, $p.x_1$ and $p.x_2$ are the x_1 and x_2 attribute values of p, and $p.t$ is its arrival time. Similar to existing approaches that handle streams of multidimensional points (e.g., [19,

31, 29, 23]), we use a regular grid to index the valid records, since a more complicated access method (e.g., a main memory R-tree) is very expensive to maintain dynamically. The extent of each cell on every dimension is δ, so that cell $c_{i,j}$ at column i and row j (starting from the low-left corner of the workspace) contains all tuples with x_1 attribute in the range $[i \cdot \delta, (i+1) \cdot \delta)$ and x_2 attribute in the range $[j \cdot \delta, (j+1) \cdot \delta)$. Conversely, given a record p with attributes $(p.x_1, p.x_2)$, its covering cell can be determined (in constant time) as $c_{i,j}$, where $i = \lfloor p.x_1/\delta \rfloor$ and $j = \lfloor p.x_2/\delta \rfloor$.

Furthermore, it is important to provide an efficient mechanism for evicting expiring records. In both versions of the sliding window (i.e., count-based and time-based), the tuples are evicted in a first-in-first-out manner, since W contains the most recent ones. Therefore, all the valid records are stored in a single list. The new arrivals are placed at the end of the list, and the tuples that fall out of the window are discarded from the head of the list. Each cell contains a list of pointers to the corresponding (valid) records, as shown in Figure 4. Since insertions and deletions to a cell also occur in a fist-in-first-out fashion, each operation on the point list takes $O(1)$ time.

The running queries q are stored in a query table QT. QT maintains for each q a unique identifier $q.id$, its scoring function $q.f$, the number of tuples required $q.k$, and its current result $q.top\_list$. The score of the kth point in $q.top\_list$ (referred to as $q.top\_score$) implicitly defines the influence region of q. To restrict the scope of the top-k maintenance algorithms, each cell c is associated with an *influence list* IL_c that contains an entry for each query q whose influence region intersects c. Since the query influence regions change dynamically, IL_c is organized as a hash-table on the query ids for supporting fast search, insertion and deletion operations.

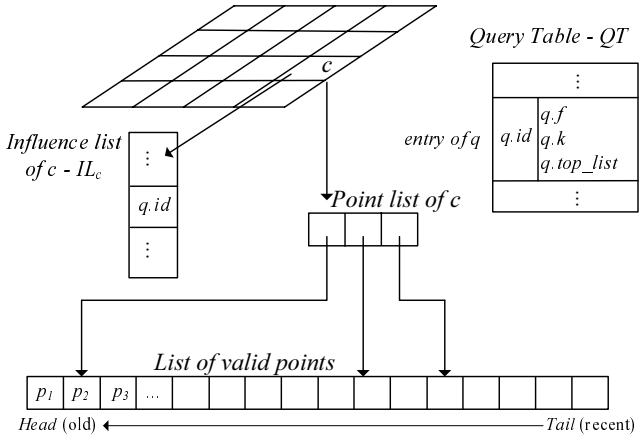
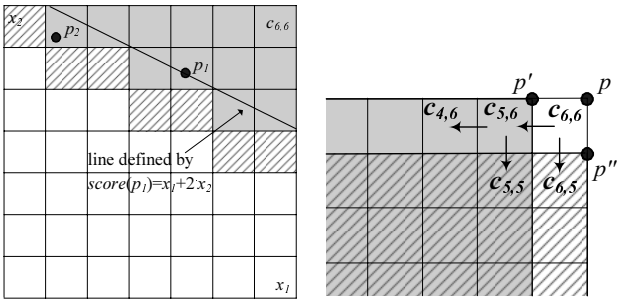


Figure 4: Index and book-keeping structures

4.2 The Computation Module

A naïve way to obtain the result of a query q is to sort all cells c according to $maxscore(c)$, and process them in descending $maxscore(c)$ order. For each visited cell, we compute $score(p)$ for every point p inside, and update $q.top\_list$ accordingly. The search terminates when the cell c under consideration has $maxscore(c) \leq q.top\_score$ (i.e., the score of the kth element in $q.top\_list$). Figure 5(a) illustrates the processing of a top-1 query q with $f(x_1, x_2) = x_1 + 2 \cdot x_2$ in a 7×7 grid. The algorithm processes the shaded cells, encounters two points p_1, p_2, and returns p_1 as the result. It can be easily shown that it is optimal in the sense that it considers only the cells intersecting the influence region. As discussed in Section 3.1, these cells must be visited anyway in order to avoid false misses. Nevertheless, it may be very expensive in practice because it requires computing the $maxscore$ for all cells and subsequently sorting them.



(a) Processed cells and points (b) Cell visiting order

Figure 5: A top-k computation example

Figure 5(b) illustrates how to determine the visiting order without computing the $maxscore$ of all cells. Since point (1,1) maximizes any function f (increasingly monotone on both dimensions), the top-right cell (in this case $c_{6,6}$) has the highest $maxscore$ and is processed first. Consider now points p' and p'' (i.e., the top-left and low-right corners of $c_{6,6}$, respectively). Point p' (p'') has higher score than any point in the shaded (striped) region. In other words, for all the unprocessed cells c, $maxscore(c) \leq \max(score(p'), score(p''))$. It follows that the cell to be visited after $c_{6,6}$ is either $c_{5,6}$ or $c_{6,5}$. Assuming that $score(p') \geq score(p'')$, $c_{5,6}$ is processed second. Similarly, the cell with the third highest $maxscore$ is determined among $c_{6,5}, c_{4,6}$, and $c_{5,5}$.

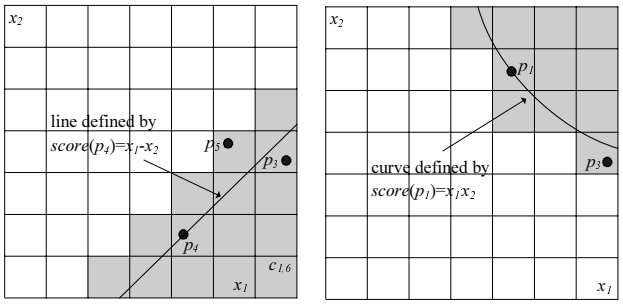
The TMA top-k computation module, shown in Figure 6, uses the above method to visit the cells in descending $maxscore$ order, preserving the property of processing the minimal set of cells. Initially, it inserts into an empty max-heap H the cell in the top-right corner of the workspace with its $maxscore$ as the sorting key. Then, it starts de-heaping cells iteratively. For each de-heaped cell $c_{i,j}$, it examines the points inside and updates $q.top\_list$. It also en-heaps $c_{i-1,j}$ and $c_{i,j-1}$ along with their $maxscore$, provided that they have not been en-heaped before. An entry for q is inserted in the influence list of processed cells, to be used for the handling of point arrivals and expirations therein. The process terminates when the next entry in H has key lower than or equal to $q.top\_score$. Note that the algorithm may en-heap some cell c even if $maxscore(c) \leq q.top\_score$ (this condition is not tested at lines 9 and 11). Such cells are not processed (i.e., not de-heaped) and therefore could be eliminated without being inserted in H. The reason for their insertion will become apparent in Section 4.3. Continuing the example of Figure 5(a), the algorithm computes the $maxscore$ and en-heaps the shaded and the striped cells. It processes (de-heaps) only the shaded ones.

In practice, there are multiple queries in the system, each with a different preference function and arbitrary value of k. The algorithm of Figure 6 can answer any query, provided that its function f is monotone on each axis. Consider the

Top-k Computation (q)
//q: top-k query
1. $q.top\_score = -\infty$; $q.top\_list = NULL$;
2. Initialize an empty max-heap H
3. Let c be the cell in the top-right corner of the workspace
4. Insert $<c, maxscore(c)>$ into H
5. While next entry has key $> q.top\_score$ and H not empty
6. De-heap the next entry $<c_{i,j}, maxscore(c_{i,j})>$ of H
7. For each point p in $c_{i,j}$
8. If $score(p) > q.top\_score$, update $q.top\_list$
9. If $c_{i-1,j}$ has not been en-heaped before
10. En-heap $<c_{i-1,j}, maxscore(c_{i-1,j})>$
11. If $c_{i,j-1}$ has not been en-heaped before
12. En-heap $<c_{i,j-1}, maxscore(c_{i,j-1})>$
13. Insert an entry for q into the influence list of $c_{i,j}$
14. End while

Figure 6: The top-k computation module

example of Figure 7(a), where $f(x_1, x_2) = x_1 - x_2$ (i.e., f is increasingly monotone on x_1 and decreasingly monotone on x_2). A top-2 search starts with the cell in the bottom-right corner of the workspace, and it en-heaps cell $c_{i,j+1}$ (instead of $c_{i,j-1}$) in lines 11-12. It returns p_3 and p_4 as the result, after processing the cells that intersect or lie below line $score(p_4) = x_1 - x_2$. Figure 7(b), on the other hand, shows a top-1 computation example for the non-linear function $f(x_1, x_2) = x_1 \cdot x_2$, which is increasingly monotone on both axes. The top-k retrieval algorithm visits the shaded cells, encounters points p_1 and p_3, and returns p_1. The influence region of the query is defined by curve $score(p_1) = x_1 \cdot x_2$. Finally, the presented method trivially extends to higher dimensionality. For instance, in 3D space the only change is that after processing $c_{i,j,w}$, cells $c_{i-1,j,w}, c_{i,j-1,w}, c_{i,j,w-1}$ are inserted into H (if they have not been en-heaped before).



(a) $f(x_1, x_2) = x_1 - x_2$, $k = 2$ (b) $f(x_1, x_2) = x_1 \cdot x_2$, $k = 1$

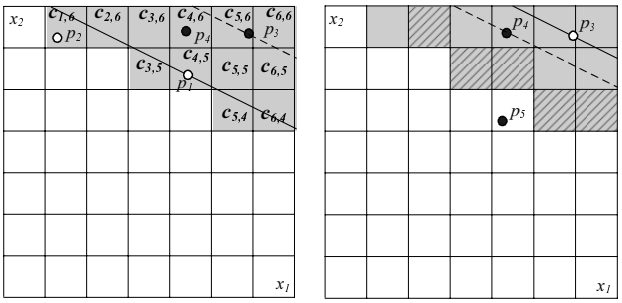
Figure 7: Top-k computation for alternative functions

4.3 The Maintenance Module

After the computation of the initial result, new records arrive at the system, while others expire. Let P_{ins} be the set of incoming tuples and P_{del} be the set of evicted ones. TMA processes P_{ins} first. For each $p \in P_{ins}$, it initially inserts (a pointer to) p into the point list of the corresponding cell c. Then, it scans the influence list IL_c of c and updates the result of every query $q \in IL_c$ for which $score(p) \geq q.top\_score$. Regarding the expiring points, for each $p \in P_{del}$, TMA deletes p from its cell c. The expunged point p may be part of the result for some of the queries in IL_c. For each query q in IL_c, if $p \in q.top\_list$, q is marked as affected, implying that its result has to be computed from scratch when the processing of P_{del} is completed.

Returning to the example of Figure 5(a), assume that tuples p_3 and p_4 arrive at the system, and at the same time p_1 and p_2 expire, as illustrated in Figure 8(a) (invalid points appear hollow). $P_{ins} = \{p_3, p_4\}$ is processed first. Cell $c_{5,6}$ of p_3 has an entry for q in its influence list, and therefore $score(p_3)$ is compared against $q.top\_score$. Since $score(p_3) > q.top\_score$, p_3 becomes the result of q. Even though $q.top\_score$ changes after the insertion of p_3, we do not update the influence lists of the cells that no longer influence q (i.e., $c_{1,6}, c_{2,6}, c_{3,6}, c_{3,5}, c_{4,5}, c_{5,5}, c_{5,4}, c_{6,4}$). The influence lists are updated only after a top-k computation from scratch, as discussed later. This "lazy" approach does not affect the correctness of the algorithm because potential insertions (or deletions) in these cells will simply be ignored (upon comparison with $q.top\_score$).

The insertion of p_4 is handled similarly, but it does not incur any change because $score(p_4)$ is lower than the new $q.top\_score$. Next, TMA processes the expiration of p_1, which no longer belongs to the result of q. Hence, it is simply removed from the point list of its cell. The maintenance algorithm proceeds with p_2, deletes it from $c_{1,6}$, and terminates. Note that if P_{del} were processed before P_{ins}, q would be marked as affected and its result would have to be computed from scratch (despite the insertion of p_3). This is the reason for handling P_{ins} before P_{del}.



(a) $P_{ins} = \{p_3, p_4\}$, $P_{del} = \{p_1, p_2\}$ (b) $P_{ins} = \{p_5\}$, $P_{del} = \{p_3\}$

Figure 8: Handling of updates

Consider now that in the next timestamp $P_{ins} = \{p_5\}$ and $P_{del} = \{p_3\}$, as shown in Figure 8(b). Processing begins with p_5, which does not cause any change. On the other hand, the deletion of p_3 invalidates the current result of q. TMA invokes the top-k computation module (Figure 6) that returns p_4. The new influence region of q contains all cells that intersect or lie above the line $score(p_4) = x_1 + 2 \cdot x_2$ (dashed in Figure 8(b)). Therefore, q must be removed from the IL_c of all cells ($c_{1,6}, c_{2,6}, c_{3,5}, c_{4,5}, c_{5,4}, c_{6,4}$) that no longer influence q (recall from Figure 8(a) that the lists of these cells were not updated during the insertion of p_3). The updating of the influence lists starts with the cells (striped in Figure 8(b)) that remain[3] in H after the termination of the top-k computation and continues in a way similar to Figure 5(b). The difference is that the order of updating the influence lists does not matter. Therefore, instead of a heap,

[3]As discussed in Figure 6, these are the cells that were en-heaped, even though their $maxscore$ was below $q.top\_score$.

we use a *list* which initially contains the cells remaining in H. For each cell $c_{i,j}$ in *list*, if $q \in IL_{c_{i,j}}$ we delete q from $IL_{c_{i,j}}$ and add $c_{i-1,j}$, $c_{i,j-1}$ into *list*, provided that they have not been inserted before. The process terminates when *list* is empty. Figure 9 presents the complete TMA algorithm.

Algorithm TMA
1. In every processing cycle do
2. P_{ins}=set of arriving points; P_{del}=set of expiring points
3. For every point p in P_{ins}
4. Insert p into the point list of the corresponding cell c
5. For each q in IL_c
6. If $score(p) \geq q.top\_score$
7. Insert p into $q.top\_list$
8. For every point p in P_{del}
9. Delete p from the point list of the corresponding cell c
10. For each q in IL_c
11. If $p \in q.top\_list$ mark q as affected
12. For each affected query q
13. Invoke Top-k Computation (q)
14. Insert the cells remaining in H into an empty *list*
15. Repeat
16. Remove next cell $c_{i,j}$ from *list*
17. If $q \in IL_{c_{i,j}}$
18. Delete q from $IL_{c_{i,j}}$
19. If $c_{i-1,j}$ is not in *list*, append $c_{i-1,j}$ to *list*
20. If $c_{i,j-1}$ is not in *list*, append $c_{i,j-1}$ to *list*
21. Until *list* is empty
22. Report changes to the client

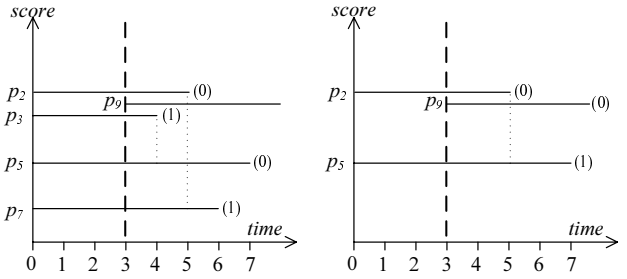
Figure 9: The TMA algorithm

In summary, the only case that involves computation from scratch for a query occurs when some of the existing top-k tuples expire and the new arrivals have a lower score than the expired records (so that the influence region expands). When a query q is terminated, we delete it from the query table, and remove q from all the influence lists in the grid. The latter task is performed by initializing *list* to contain the corner cell with the maximum *maxscore*.

5. SKYBAND MONITORING ALGORITHM

The Skyband Top-k Monitoring Algorithm (SMA) applies the reduction from top-k to k-skyband queries in order to avoid computation from scratch when some results expire. Consider, for instance, Figure 10(a), where tuples are shown as intervals in the 2-dimensional *score-time* space. The number in the parenthesis corresponds to the *dominance counter* (DC) of each tuple p, i.e., the number of records with higher score that arrive after[4] p. At time 0, the result of a top-2 query contains p_2 and p_3, whereas the 2-skyband contains p_2, p_3, p_5, p_7. At time 3, p_9 arrives, and expires after all other records in the system. It follows that (i) p_9 is not dominated by any point (i.e., $p_9.DC = 0$), and (ii) all the points p with $score(p) \leq score(p_9)$ are dominated by p_9. Therefore, the dominance counters of p_5, p_3, p_7 increase by one, i.e., $p_5.DC = 1$ and $p_3.DC = p_7.DC = 2$. Consequently, p_3 and p_7 are removed from the 2-skyband at time 3. The updated 2-skyband, shown in Figure 10(b), contains p_2, p_9 and p_5. The new top-2 set consists of the two elements in the skyband with the highest scores (i.e., p_2 and p_9). After the expiration of p_2 (at time 5) the top-2 result will change to $\{p_5, p_9\}$.

[4]In both count-based and time-based windows the arrival order is the same as the expiration order.



(a) Point p_9 arrives (b) New 2-skyband

Figure 10: Skyband maintenance

In general, the monitoring of future top-k results reduces to a k-skyband maintenance task. SMA restricts the skyband maintenance for a query q to points falling inside its influence region. Specifically, the initial top-k results of q (and their scores) are retrieved by the top-k computation module of Figure 6 and are inserted into $q.skyband$, which contains entries of the form $<p.id, p.score, p.DC>$ in descending order of $p.score$. Then, SMA scans $q.skyband$, and for each tuple p it computes $p.DC$. To speed up the dominance counter computation, the arrival time of every processed element of $q.skyband$ is stored into a balanced tree BT sorted in descending order. Thus, $p.DC$ is simply the number of tuples that precede p in BT (since records are processed in descending score order, these tuples are preferable over p in terms of both score and expiration time). An internal node in BT contains the cardinality of the sub-tree rooted at that node so that the computation of dominance counters takes in total $O(k \cdot \log k)$ time. After the dominance counter computation, BT is discarded and the $q.skyband$ contains exactly k tuples; $q.top\_score$ is the score of the kth tuple.

The skyband maintenance procedure only handles tuples p with $score(p) \geq q.top\_score$. When such a tuple arrives at the system, it is inserted into $q.skyband$ increasing its cardinality. The first k records of the skyband constitute the $q.top\_list$ (in accordance with the TMA terminology), which is not stored explicitly. The dominance counter of all records with score lower than $score(p)$ is increased by 1 and the tuples whose counter reaches k are evicted. Regarding deletions, the element p of $q.skyband$ with the earliest arrival time (i.e., the one expiring first) belongs[5] to the current result. Hence, when p expires, it is removed and the first k elements of $q.skyband$ are reported as the new $q.top\_list$. Note that p does not dominate any point, and therefore the dominance counters of the remaining elements in $q.skyband$ are not affected. The SMA algorithm is illustrated in Figure 11.

An important remark concerns the situation where the skyband contains fewer than k points. This scenario occurs when some top-k results expire and the recent arrivals were not inserted in the skyband (because their score was below $q.top\_score$). In such cases, we have to re-apply the algo-

[5]This can be proven by contradiction. Assume that the expiring point p is not in the current top-k result. Then, all the tuples in the result dominate p since they have higher score and expire later. Thus, p cannot belong to the k-skyband.

Algorithm SMA
1. In every processing cycle do
2. P_{ins} = set of arriving points
3. P_{del} = set of expiring points
4. For every point p in P_{ins}
5. Insert p into the point list of the corresponding cell c
6. For each q in IL_c
7. If $score(p) \geq q.top\_score$ *//score of the kth element after the last application of top-k computation (q)*
8. Insert p into $q.skyband$ and set $p.DC = 0$
9. For each p' in $q.skyband$ with $score(p') \leq score(p)$
10. $p'.DC = p'.DC + 1$
11. If $p'.DC = k$ evict p' from $q.skyband$;
12. For every point p in P_{del}
13. Delete p from the point list of the corresponding cell c
14. For each q in IL_c
15. If $p \in q.top\_list$
16. Delete p from $q.skyband$
17. For each query q whose skyband has changed
18. If $q.skyband$ has at least k points
19. $q.top\_list$ = the first k elements of $q.skyband$
20. Else *//q.skyband has fewer than k points*
21. Invoke Top-k Computation (q)
22. Form $q.skyband$ and compute dominance counters
23. Report changes to the client

Figure 11: The SMA algorithm

rithm of Figure 6 and compute the skyband from scratch. The pseudo-code of Figure 11 handles this case in lines 20-22. SMA is expected to be faster than TMA, since it involves less frequent calls to the top-k computation module. For example, in Figure 8(b), SMA would have kept point p_4 in the skyband, and report it as the result when p_3 expires. On the other hand, the space requirements of SMA are higher than TMA, since it maintains the skyband (which is a superset of the current top-k set) of each query. In the next section we analytically compare the running time and space requirements of the proposed algorithms.

6. PERFORMANCE ANALYSIS

Similar to previous analytical studies in the literature [18, 23, 29, 31], we assume that (i) the average data cardinality at each timestamp is N, (ii) the tuples are uniformly distributed in a unit d-dimensional workspace, and (iii) the stream rate is, on the average, r tuples per processing cycle. If δ is the cell extent per axis, the total number of cells is $(1/\delta)^d$ and each cell contains on the average $N \cdot \delta^d$ points. First, we analyze the running time of the top-k computation module (involved in both TMA and SMA). Recall that the algorithm visits only the cells intersecting the influence region of a query. The influence region contains k (out of the N) records and, according to the uniformity assumption, has volume k/N. Thus, the number of processed cells is $C = O(\lceil k/(N \cdot \delta^d) \rceil)$. Each cell is inserted into (deleted from) a heap with logarithmic cost, resulting in $O(C \cdot \log C)$ cost for heap operations. The number of points in the processed cells is $|C| = O(C \cdot N \cdot \delta^d)$. Each of these points is considered for insertion into the $q.top\_list$ (or $q.skyband$). With a red-black tree implementation, an update of the $q.top\_list$ costs $O(\log k)$ time (i.e., a deletion of its kth element, followed by an insertion of a new one). Thus, the running time of the top-k search is $T_{comp} = O(C \cdot \log C + |C| \cdot \log k)$.

Concerning the maintenance cost of TMA, in every processing cycle, r new tuples arrive at the system, while r old ones expire. Hence, the grid update time is $O(r)$. Each cell receives $r \cdot \delta^d$ insertions and $r \cdot \delta^d$ deletions. Therefore, the influence region of a top-k query q is affected by $2 \cdot C \cdot r \cdot \delta^d$ events. The time required to check whether the corresponding points belong to the current result is $O(C \cdot r \cdot \delta^d)$ (by comparing with $q.top\_score$). Among them, $k \cdot r/N$ new points are considered for insertion into $q.top\_list$, and $k \cdot r/N$ old ones are deleted from it, with cost $O(k \cdot r \cdot \log k/N)$. Let Pr_{rec} be the probability that the query has to be processed from scratch after the updates. It holds that $Pr_{rec} \leq 1 - (1 - (r/N))^k$, where $(1 - (r/N))^k$ is the probability that none of the current top-k entries expire. The quantity $1 - (1 - (r/N))^k$ is an upper bound of Pr_{rec}, as some arriving points may replace expiring entries. Summarizing, the time complexity of TMA for a processing cycle is $T_{TMA} = O(r + Q \cdot (C \cdot r \cdot \delta^d + k \cdot r \cdot \log k/N + Pr_{rec} \cdot T_{comp}))$, where Q is the number of running queries.

For SMA, the index update cost is the same as for TMA (i.e., $O(r)$). Also, the number of the arriving (expiring) points in the cells intersecting the influence region of a query q is $O(C \cdot r \cdot \delta^d)$. Initially (after the application of the top-k computation module), the skyband contains k elements. Among the inserted (deleted) points, $O(k \cdot r/N)$ have score greater than or equal to $q.top\_score$ and have to be included in (excluded from) the skyband. An insertion to $q.skyband$ requires $O(k)$ time, because we have to retain the order (according to score), and at the same time update the dominance counters of the entries with score lower than that of the new record. Similarly, each deletion has $O(k)$ cost. Note that for uniform data distribution, the number of insertions in the influence region of q equals the number of deletions therein, and the k-skyband contains exactly k entries. Thus, SMA does not resort to top-k computation from scratch (this observation is also verified in the experiments). In this case the total running time is $T_{SMA} = O(r + Q \cdot (C \cdot r \cdot \delta^d + k^2 \cdot r/N))$ for each processing cycle.

Finally, we analyze the memory requirements of the proposed methods. The index has $O(N \cdot d + N + Q \cdot C)$ size, where $O(N \cdot d)$, $O(N)$ and $O(Q \cdot C)$ are the amounts of storage required for the N valid d-dimensional points, for N pointers (in the point lists of the cells), and for the influence lists of the Q queries. Each query table entry for TMA has size $O(d + 2 \cdot k)$, for storing the scoring function parameters and the tuple $<p.id, score(p)>$ for every point p in the result. For SMA each entry of QT takes up $O(d + 3 \cdot k)$, since in addition to the id and the score, $q.skyband$ also contains the dominance counters of the points. Recall that SMA does not need to explicitly store $q.top\_list$, because the result set consists of the first k entries of $q.skyband$. Summarizing, the space requirements of TMA and SMA are $S_{TMA} = O(N \cdot (d + 1) + Q \cdot (C + d + 2 \cdot k))$ and $S_{SMA} = O(N \cdot (d + 1) + Q \cdot (C + d + 3 \cdot k))$, respectively.

In general, SMA is expected to be faster than TMA because the latter one resorts more frequently (with probability Pr_{rec}) to top-k computation from scratch. On the other hand, the result updating of TMA is more efficient than the skyband maintenance of SMA (with time complexities $O(k \cdot r \cdot \log k/N)$ and $O(k^2 \cdot r/N)$ per query, respectively). Therefore, if Pr_{rec} is very small, TMA outperforms SMA. As shown in the experimental evaluation, however, this case is rare. Concerning the space overhead, SMA uses more memory than TMA because (i) the $q.skyband$ stores additional information about the dominance counters, and

(ii) in practice, the k-skyband may contain more than k entries. The performance of both algorithms depends on the cell side-length δ. Large cells minimize the time spent on heap operations, but lead to unnecessary processing of points that are outside the influence region (but fall in cells that intersect the influence region). Large δ also implies lower space consumption, because queries are affected by fewer cells, and the cell influence lists take up less memory. The running time of the proposed techniques increases with k, Q, N, and r. The same holds for the space consumption, with the exception of r.

7. OTHER QUERY TYPES AND STREAM MODELS

In this section we discuss the extension of the presented algorithms to special cases of top-k monitoring, as well as their adaptation to another data stream model. A *constrained* top-k query q monitors only points falling in the sub-space defined by a set of input constraints. Typically, each constraint is expressed as a range along a dimension and the conjunction of all constraints forms a hyper-rectangle (referred to as the *constraint region*) in the d-dimensional attribute space. Figure 12 illustrates an example of a top-1 query q with $f(x_1, x_2) = x_1 + 2 \cdot x_2$, where the constraint region is an axis-parallel rectangle R. To compute the initial result of q, the top-k computation module starts with cell $c_{5,5}$ that maximizes f in R. Then, it proceeds with $c_{4,5}$ where it encounters p_1. Since p_1 does not lie in R, it is excluded from consideration. Finally, p_2 is returned as the result. The visited cells (appearing shaded) receive an entry for q in their influence lists. Among the point insertions and deletions in these cells, only the ones falling in R are processed by the maintenance algorithm.

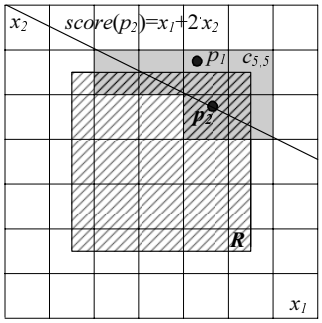


Figure 12: A constrained top-1 query example

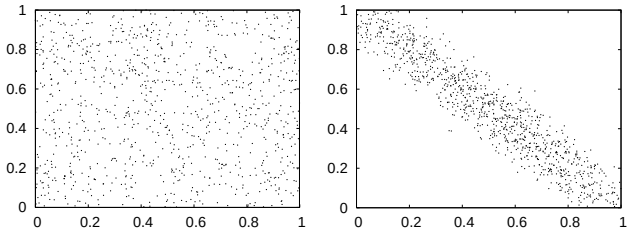
Another interesting type of preference-based retrieval concerns queries that request monitoring of all the points with score above a user-specified *threshold*. A simple, but expensive, method is to check each arriving/expiring entry against all the queries. On the other hand, the proposed framework provides an efficient and scalable processing technique. Given a new threshold query q, the initial result computation starts with the cell at the top-right corner of the workspace, and proceeds with the adjacent cells provided that their *maxscore* is higher than *threshold*. Since the visiting order is not important, the search can be performed with a list (instead of a heap), similar to the way discussed in Section 4.3. An entry for q is inserted into the influence list of each processed cell. The maintenance module simply reports the point insertions and deletions in these cells for points p with $score(p) > threshold$.

So far we have assumed the append-only data stream model. In case of streams that contain explicit deletions, the data no longer expire in a first-in-first-out manner. Therefore, we do not maintain the valid data list described in Section 4.1. Instead, when a tuple p arrives at the system, it is directly placed into its cell in the grid. When a deletion update is issued for p, it is deleted from the corresponding cell. The point lists of the cells are implemented as hash-tables for supporting random insertions/deletions in constant expected time. TMA applies directly to this scenario; when some of the current top-k points of a query q are deleted, then its result is computed from scratch. On the other hand, the skyband computation and maintenance of SMA is not possible because the expiry order of the tuples is not known in advance.

8. EXPERIMENTAL EVALUATION

In this section we experimentally evaluate the proposed methods using streams of *independent* (IND) and *anti-correlated* (ANT) data distributed in a unit workspace of dimensionality d in the range from 2 to 6. For IND data, the attribute values of each tuple are generated independently, following a uniform distribution. ANT data are generated in the way described in [4], and represent an environment where points that have a large value on one dimension, have small values on one or all of the remaining dimensions. IND and ANT are common benchmarks for preference-based queries [4, 24]. Figure 13 illustrates two example datasets.

We assume a count-based window with size N between 1 and 5 million tuples. During each timestamp, r new points arrive at the system. We generate Q top-k monitoring queries with scoring functions of the form $f(p) = \sum_{i=1}^{d} a_i \cdot p.x_i$, where the a_i coefficients are randomly chosen between 0 and 1. The simulation length is 100 timestamps. We compare the performance of three algorithms: (i) *threshold sorted list* (TSL), that combines the TA algorithm with the maintenance module of [30] (as discussed in Section 3.2), (ii) TMA, and (iii) SMA. Table 1 summarizes the parameters under investigation, along with their ranges and default values. In each experiment we vary a single parameter, while setting the remaining ones to their default values. For all simulations we use a Pentium 3.2 GHz CPU with 1 GByte memory.



(a) IND (b) ANT

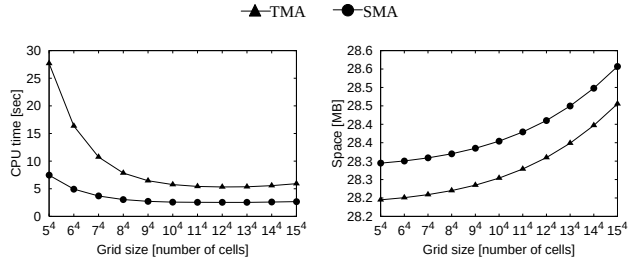
Figure 13: Datasets ($d = 2$)

First, we study the effect of the grid granularity on TMA and SMA for IND and the default settings (i.e., $d = 4$, $N = 1$M, $r = 10$K, $Q = 1$K, $k = 20$). Each axis is divided into a number of equal intervals that varies between 5 and

Table 1: System parameters

Parameter	Default	Range
Data dimensionality (d)	4	2,3,4,5,6
Data cardinality (N)	1M	1,2,3,4,5 (M)
Arrival rate (r)	10K	1,5,10,50,100 (K)
Query cardinality (Q)	1K	100,500,1K,2K,5K
Result cardinality (k)	20	1,5,10,20,50,100

15 (corresponding to a grid of 5^4 up to 15^4 cells). Figure 14 illustrates the overall running time of the methods as well as their space requirements. A grid of 12^4 cells is the best choice in terms of speed for both algorithms. A finer grid is more expensive because of the heap operations on the cells (some of which are empty), whereas a sparser grid leads to unnecessary processing of points outside the query influence regions (as discussed in Section 6). Regarding the memory footprints, a finer grid results in higher space consumption, mainly due to the book-keeping. The diagrams for ANT follow the same trends and are omitted.



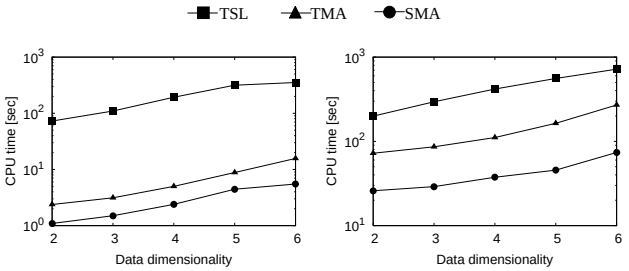
(a) CPU time (b) Space

Figure 14: Performance vs. grid granularity (IND)

For fairness, we also fine-tune the value of k_{max} in the TSL algorithm, for different values of k. Using the default settings and the IND dataset, we identified the optimal values $(4, 10, 20, 30, 70, 120)$ for k_{max}, corresponding to the values $(1, 5, 10, 20, 50, 100)$ of k. Note that in the original paper [30] the authors propose an algorithm that dynamically adjusts the value of k_{max}, according to the ratio of the refill operation cost over the tuple update cost. However, this approach performs worse than TSL with fine-tuned k_{max}.

In Figure 15 we set N = 1M and r = 10K, and measure the running time for different data dimensionalities d (ranging between 2 and 6). The cell extent is selected so that the grid contains approximately 12^4 cells because, as shown in Figure 14, this value provides the best performance with reasonable space requirements. The cost of all algorithms increases with d. For SMA and TMA, this happens because more cells are processed during the top-k computation from scratch (recall that after de-heaping a cell, we en-heap d adjacent ones). For TSL, this is because the number of lists and the cost of TA computations is proportional to d. The significant gain of TMA over TSL demonstrates the benefits of the top-k computation module compared to the TA algorithm (note that TMA maintains exactly k results and, thus, performs more top-k computations than TSL). The advantage of SMA over TMA, is due to the less frequent re-computations (of the top-k result) from scratch.

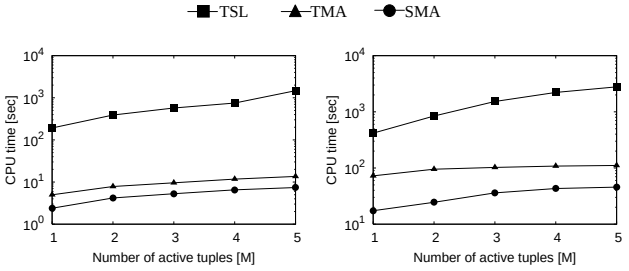
In Figure 16 we vary N from 1M to 5M, and set the arrival rate r to $N/100$ tuples per timestamp. In other words, during each timestamp, 1% of the data points are replaced



(a) IND (b) ANT

Figure 15: CPU time vs. d

by new ones. As expected, the performance of all algorithms degrades with N. However, both our methods scale much better than TSL, and they are more than one order of magnitude faster in most cases. An interesting observation, which is apparent in all experiments, is that the cost increases for anti-correlated data. This happens because for ANT, the data are concentrated close to the plane that passes through point (0.5,0.5,0.5,0.5) and is perpendicular to the line crossing (0,0,0,0) and (1,1,1,1). Thus, the top-k computation module (that starts from the top-right corner of the data space) has to process many cells before retrieving k results.



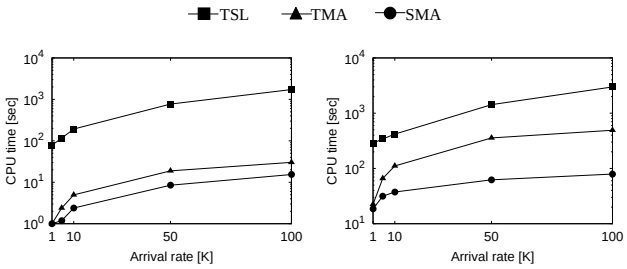
(a) IND (b) ANT

Figure 16: CPU time vs. N ($r = N/100$)

Next, we set N = 1M and vary r between 1K and 100K, i.e., 0.1% up to 10% of the valid tuples are replaced per timestamp. As shown in Figure 17, the cost of TMA and SMA increases with r, verifying the analysis of Section 6. For all values, both our monitoring algorithms outperform TSL, showing better resilience against frequent updates. This is due to the high update cost of TSL that involves the on-line maintenance of d sorted lists. Furthermore, SMA performs significantly better than TMA for anti-correlated data, since the cost of the (frequent) top-k computations is higher (as explained in the context of Figure 16). Figure 18 measures the effect of the query cardinality, ranging between 100 and 5K. The running time of all methods scales linearly with Q, while their relative performance is similar to the previous experiments.

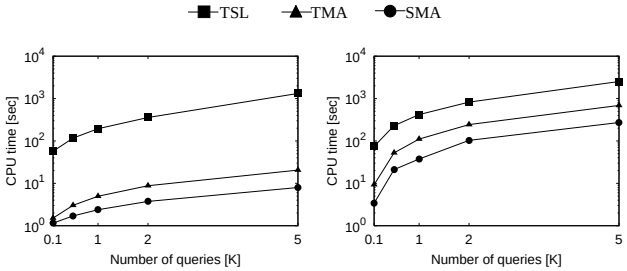
Figure 19 shows the running time versus k for IND and ANT. The influence regions of the queries and, consequently, the number of processed cells grow with k, implying higher result computation and maintenance overhead. Initially, the costs of TMA and SMA are similar, but their performance gap increases with k. This happens because high values of k raise the probability Pr_{rec} that some of the current top-k

points expire. For $k = 100$ and ANT, the cost of TMA is almost as high as that of TSL, due to the very frequent and expensive re-computations of top-k results from scratch.



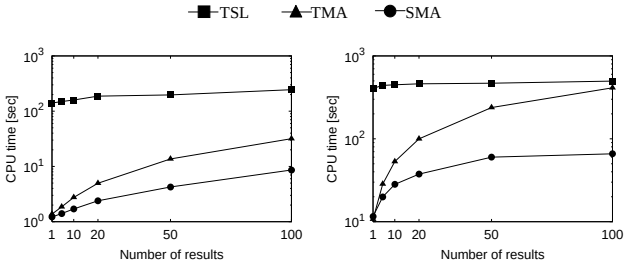
(a) IND (b) ANT

Figure 17: CPU time vs. r



(a) IND (b) ANT

Figure 18: CPU time vs. Q

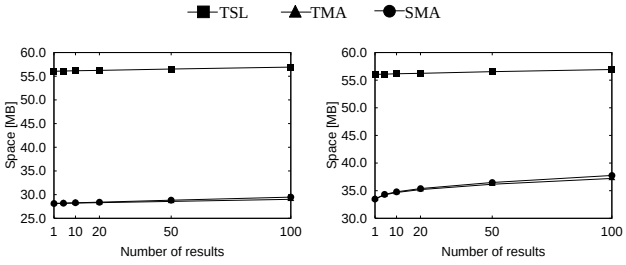


(a) IND (b) ANT

Figure 19: CPU time vs. k

Figure 20 displays the space overhead for the same experiment. TMA and SMA need to maintain for each cell (i) the influence lists, and (ii) the corresponding point list. The overhead of the algorithms increases with k because they have to store more result tuples per query. Moreover, for TMA and SMA, the influence lists grow with k. TSL consumes more space compared to our methods due to the additional d sorted lists. Table 2 shows the average cardinality of the materialized views and of the skybands, for TSL and SMA, respectively. In accordance with the analysis of Section 6, SMA maintains very few extra points. Note that SMA maintains fewer points in the top-k view than TSL, since it continuously discards those tuples that can never appear in the result.

Finally, we demonstrate the effectiveness of the proposed algorithms for non-linear functions. Figures 21(a) and 21(b) evaluate the cost of queries with $score(p) = \prod_{i=1}^{d}(a_i + p.x_i)$, where $a_i \in [0, 1]$, as a function of d for IND and ANT. Figures 21(c) and 21(d), repeat the experiment for 1K queries with $score(p) = \sum_{i=1}^{d} a_i \cdot p.x_i^2$. The relative performance of the algorithms is similar to the case of linear functions (Figure 15), illustrating the generality of our methods.

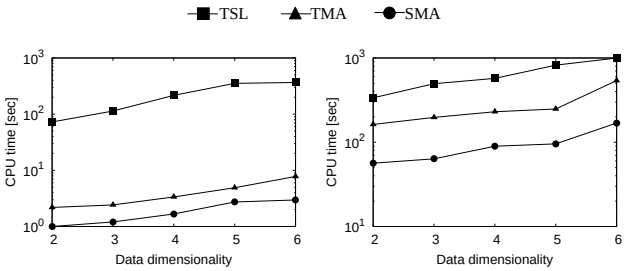


(a) IND (b) ANT

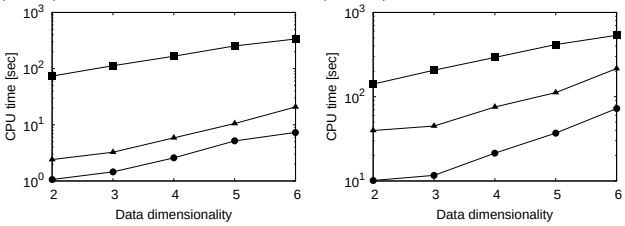
Figure 20: Space requirements vs. k

Table 2: Average view/skyband size per query

k	IND		ANT	
	TSL	SMA	TSL	SMA
1	3.3	1.1	3.1	1.1
5	8.6	5.9	8.4	5.9
10	17.1	11.2	17.2	11.5
20	26.7	21.6	26.9	22.4
50	63.0	53.3	64.4	54.4
100	113.2	104.6	113.6	106.5



(a) $f(p) = \prod_{i=1}^{d}(a_i + p.x_i)$ (IND) (b) $f(p) = \prod_{i=1}^{d}(a_i + p.x_i)$ (ANT)



(c) $f(p) = \sum_{i=1}^{d} a_i \cdot p.x_i^2$ (IND) (d) $f(p) = \sum_{i=1}^{d} a_i \cdot p.x_i^2$ (ANT)

Figure 21: CPU time vs. d for non-linear f

9. CONCLUSIONS

This paper constitutes the first work on continuous monitoring of top-k queries over sliding windows. We use a regular grid to index the valid records in main memory. The initial result of each query is computed by a top-k computation module that processes the minimum number of cells.

Only record updates within these cells can potentially invalidate the current top-k set. Therefore, the maintenance of the result considers only point arrivals and expirations therein. We propose two monitoring algorithms, TMA and SMA, that differ in the way that they handle expirations of top-k records. TMA re-computes the result from scratch, whereas SMA maintains a superset of the current answer in the form of a k-skyband, in order to avoid frequent re-computations. An extensive experimental evaluation illustrates that incremental result maintenance pays off for both TMA and SMA. Moreover, it demonstrates that SMA outperforms TMA for all parameter settings, with a small space overhead. An interesting direction for future work concerns processing queries with non-monotone preference functions. Geometric reasoning could be employed for answering several classes of such functions. For example, a function with finite and analytically computable local maxima could be evaluated with a proper partitioning of the space into subdomains where it is monotone.

10. ACKNOWLEDGMENTS

This work was supported by grant HKUST 6184/05E from Hong Kong RGC.

11. REFERENCES

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[2] B. Babcock and C. Olston. Distributed top-k monitoring. In *SIGMOD*, pages 28–39, 2003.

[3] W.-T. Balke, U. Güntzer, and J. X. Zheng. Efficient distributed skylining for web information systems. In *EDBT*, pages 256–273, 2004.

[4] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[5] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *TODS*, 27(2):153–187, 2002.

[6] Y. Cai, K. A. Hua, and G. Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *MDM*, pages 27–38, 2004.

[7] K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD*, pages 346–357, 2002.

[8] Y.-C. Chang, L. D. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The Onion technique: Indexing for linear optimization queries. In *SIGMOD*, pages 391–402, 2000.

[9] S. Chaudhuri, L. Gravano, and A. Marian. Optimizing top-k selection queries over multimedia repositories. *TKDE*, 16(8):992–1009, 2004.

[10] C.-M. Chen and Y. Ling. A sampling-based estimator for top-k query. In *ICDE*, pages 617–627, 2002.

[11] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top N queries. In *VLDB*, pages 411–422, 1999.

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

[13] B. Gedik and L. Liu. MobiEyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, pages 67–87, 2004.

[14] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *VLDB Journal*, 13(1):49–70, 2004.

[15] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Joining ranked inputs in practice. In *VLDB*, pages 950–961, 2002.

[16] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB Journal*, 13(3):207–221, 2004.

[17] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid. Rank-aware query optimization. In *SIGMOD*, pages 203–214, 2004.

[18] D. V. Kalashnikov, S. Prabhakar, and S. E. Hambrusch. Main memory evaluation of monitoring queries over moving objects. *Distributed and Parallel Databases*, 15(2):117–135, 2004.

[19] N. Koudas, B. C. Ooi, K.-L. Tan, and R. Zhang. Approximate NN queries on streams with guaranteed error/performance bounds. In *VLDB*, pages 804–815, 2004.

[20] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *ICDE*, pages 502–513, 2005.

[21] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *TODS*, 29(2):319–362, 2004.

[22] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, pages 623–634, 2004.

[23] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, pages 634–645, 2005.

[24] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *TODS*, 30(1):41–82, 2005.

[25] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *Transactions on Computers*, 51(10):1124–1140, 2002.

[26] Y. Tao and D. Papadias. Maintaining sliding window skylines on data streams. *TKDE*, 18(3):377–391, 2006.

[27] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB*, pages 648–659, 2004.

[28] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava. Ranked join indices. In *ICDE*, 2003.

[29] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654, 2005.

[30] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *ICDE*, pages 189–200, 2003.

[31] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, pages 631–642, 2005.