Ad-hoc Top-k Query Answering for Data Streams

Gautam Das
University of Texas at Arlington
gdas@cse.uta.edu

Dimitrios Gunopulos
University of California, Riverside
dg@cs.ucr.edu

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

Nikos Sarkas
University of Toronto
nsarkas@cs.toronto.edu

ABSTRACT

A top-k query retrieves the k highest scoring tuples from a data set with respect to a scoring function defined on the attributes of a tuple. The efficient evaluation of top-k queries has been an active research topic and many different instantiations of the problem, in a variety of settings, have been studied. However, techniques developed for conventional, centralized or distributed databases are not directly applicable to highly dynamic environments and on-line applications, like data streams.

Recently, techniques supporting top-k queries on data streams have been introduced. Such techniques are restrictive however, as they can only efficiently report top-k answers with respect to a pre-specified (as opposed to ad-hoc) set of queries. In this paper we introduce a novel geometric representation for the top-k query problem that allows us to raise this restriction. Utilizing notions of geometric arrangements, we design and analyze algorithms for incrementally maintaining a data set organized in an arrangement representation under streaming updates. We introduce query evaluation strategies that operate on top of an arrangement data structure that are able to guarantee efficient evaluation for ad-hoc queries. The performance of our core technique is augmented by incorporating tuple pruning strategies, minimizing the number of tuples that need to be stored and manipulated. This results in a main memory indexing technique supporting both efficient incremental updates and the evaluation of *ad-hoc* top-k queries. A thorough experimental study evaluates the efficiency of the proposed technique.

1. INTRODUCTION

The data stream model of computation [5] best captures the data and query characteristics of many modern applications, including network data management, financial data monitoring and sensor networks. A stream of tuples arrives continuously at possibly high rates and a main memory buffer maintains incoming tuples. As the memory space is limited, aged tuples are evicted in order to free space for fresh incoming tuples. Several policies for managing data in the main memory buffer have been studied. A natural policy capturing the memory constraint requirements is to consider tuples valid as long as they belong to a sliding window of a specific size W. Such a window can be *time based* or *tuple-count based*. Time based sliding windows assume that tuples arrive with a time stamp and remain in the buffer as long as their time stamp belongs to a fixed time period covering the most recent time stamps. Tuple-count based sliding windows contain the most recent N records.

A basic requirement for many monitoring applications in a stream setting is to be able to rank tuples in the buffer according to *ad-hoc* preferences towards their attributes. In a sensor application in which tuples of readings arrive continuously, it is commonly required to order the tuples in the buffer in an ad-hoc way. For example, one may be interested to report the k highest ranking tuples in the buffer according to the temperature attribute, followed by a request to report the k highest ranking tuples according to humidity (for a suitably defined value of k), followed by some ad-hoc weighted combination of both, etc. Similar requirements exist in streams with more rapid rates, such as IP network streams. There we might be interested to rank tuples (packets) by destination port or source port, or ad-hoc weighted combinations of packet length and number of network hops, etc. Depending on the application context, endless possibilities exist.

In this paper we focus on supporting efficient ad-hoc top-k query answering over the contents of such a buffer. Previous work has investigated the problem of efficiently maintaining the result of a persistent set of top-k queries as the buffer is updated. This is too restrictive as one must have a clear idea on what queries one wishes to ask before hand. Therefore, we present techniques designed to support the efficient evaluation of ad-hoc top-k queries. In particular we make the following contributions:

- We introduce a novel geometric representation of the top-k query answering problem that allows us to utilize an *arrangement* of geometric objects, in order to perform indexing and ad-hoc top-k query answering.
- We present algorithms for updating and querying such an index and study their complexity.
- We introduce and study tuple pruning methods, aiming to minimize the number of tuples that need to be indexed, while maintaining the capability to correctly answer any candidate top-k query. We saw how to efficiently implement them by utilizing query maintenance techniques [26, 19].
- Using a combination of real and synthetic data sets we evaluate the performance of our techniques for a variety of parameter settings, demonstrating its overall performance and efficiency.

The rest of the paper is organized as follows. In Section 2 we review related work. Section 3 formally defines the top-k query answering problem and presents background material necessary for the remainder of our study. In Section 4 we present our ad-hoc query evaluation methodology using arrangements. Section 5 describes tuple pruning techniques incorporated in the core arrangements solution and dynamic maintenance issues. Section 6 discusses issues related to tuning our method. In Section 7 we discuss the generalization of our techniques to high dimensions and their application to an interesting variation of top-k queries. Finally, Section 8 presents the results of our experimental evaluation; we conclude in Section 9.

2. RELATED WORK

Top-k queries were first introduced in the context of multimedia systems [12, 13]. Queries over multimedia content are rarely exact. Instead, the objects most similar to the query are to be retrieved. A multimedia system scores the objects according to how well they match each of the query predicates and produces a sorted list in descending score order for each of the predicates. These lists are subsequently combined to produce the final ranking of the objects with respect to the whole query. This final ranking is based on a monotone scoring function defined over the partial scores.

Merging partial results is performed by the Threshold Algorithm (TA) [22]. Variations of the TA algorithm exist depending on whether random accesses to the lists are allowed or prohibited. Several extensions to the basic TA algorithm have been proposed. [23] developed a version that produces approximate results, while offering probabilistic guarantees about their precision. [6] use statistics on the lists to optimize the performance of the TA algorithm, while [10] allows the TA algorithm to use results of previously answered queries in addition to the sorted lists.

The *Onion* indexing technique [9] organizes the data into layers of convex hulls and is able to answer queries using additive scoring functions by processing the layers inwards, starting from the outmost hull. The technique is therefore able to answer ad-hoc queries, however it is mostly aimed at static data, since the hulls are very expensive to maintain dynamically. Several other top-k techniques based on indices have been proposed [24, 17].

Most relevant to our problem is the top-k query monitoring techniques of Mouratidis et al., [19]. The techniques, named *TMA* and *SMA*, employ a regular grid to index the buffer and use it to perform both top-k query answering and maintenance. Although top-k query evaluation is supported, it is inefficient and the methods rely instead on the incremental maintenance of the results of a fixed set of queries in order to avoid expensive top-k recomputations. Our approach raises such a restriction being able to efficiently answer *ad-hoc* queries.

In [19], top-k computation is performed by visiting the cells of the grid in descending maximum possible tuple score (max-score) order, as determined by the score with respect to the query of the upper-right corner of the cell. Query evaluation is facilitated by a priority queue: the cell with the highest max-score is deheaped, the tuples inside it are processed and its neighboring cells are enheaped. The procedure terminates when the score of the k-th tuple in the current result is higher than the max-score of the top cell in the heap.

For result maintenance, each query is associated with a number of grid cells that constitutes its *influence region* and only updates that happen within the influence region of a query are processed. The SMA method achieves better running times over TMA by taking into account future tuple expirations that will affect queries and compensating by maintaining additional results per query.

3. BACKGROUND

We start by introducing material required for the remainder of the paper. Section 3.1 formally defines the top-k query answering problem. Section 3.2 introduces the notion of an *arrangement* of geometric objects [20, 11, 3, 16] that is utilized by our solution .

3.1 Top-k query answering

Consider a data set D of n tuples $t_1, \dots, t_n$ with d numeric attributes $X_1, \dots, X_d$. Data set D is a snapshot of our main memory buffer $\mathcal{B}$ at a specific time instance. Let Dom_i be the domain of the i-th attribute. Without loss of generality, the domain of every attribute is considered to be the unit interval $[0, 1]$. Throughout the paper we will use this convention. Each tuple can also be viewed as a numeric vector $\vec{t} = (t.X_1, \dots, t.X_d)$. A top-$k$ ranking query can be expressed as a pair $Q = (S, k)$ returning the k highest ranking (scoring) tuples with respect to a scoring function $S : Dom_1 \times \cdots \times Dom_d \to \mathbb{R}$, defined on the attributes of a tuple. To ease notation, we assume that S involves all attributes but our discussion remains valid for functions involving only a subset of the attributes. Moreover, to simplify our presentation we choose to present our framework for the case of $d = 2$. We generalize our framework in Section 7. In accordance to prior art [24, 10, 17, 9], we consider linear additive scoring functions of the form $S(t) = \vec{w}\vec{t} = w_1 t.x_1 + w_2 t.x_2$, where w_1 and w_2 are positive, real constants. Thus, a scoring function can be simply expressed as a vector $\vec{w} = (w_1, w_2)$.

Both data set D and a top-k query $Q = (\vec{w}, k)$ have natural geometric representations. Such a geometric representation utilizes the native coordinates of the tuples in D, as well as the query parameters, and we refer to it as the representation in the *primal plane*. This mapping has been extensively used in different variations of the top-k query answering problem [9, 24, 19, 10]. Each tuple $t = (x_1, x_2)$ in D corresponds to a point $p(t) = (t.x_1, t.x_2)$ that lies inside the unit square $[0, 1] \times [0, 1]$. A query Q can be mapped to the vector $\vec{w} = (w_1, w_2)$ corresponding to its scoring function. Notice that the score of t is equal to the dot product $\vec{p(t)}\vec{w} = w_1 t.x_1 + w_2 t.x_2$. This dot product is proportional (times $|\vec{w}|$) to the distance of the projection of p on $\vec{w}$, from the origin of the space (in this case point $(0, 0)$) . This establishes a connection between the ordering of the points' projections on $\vec{w}$ and the ranking of the corresponding tuples with respect to the scoring function: the order in which we meet the projections of the points as we move along the supporting line of $\vec{w}$, from infinity towards the origin, is the same as the ranking of the corresponding tuples (Figure 1).

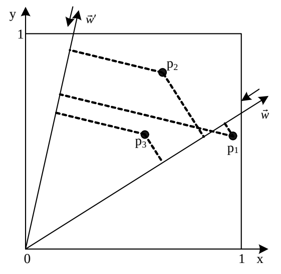


Figure 1: Top-k query answering in the primal plane.

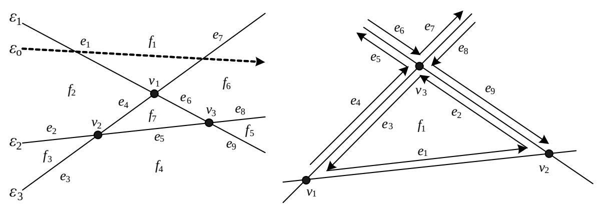
We would like to support ad-hoc top-k queries on the contents of buffer $\mathcal{B}$ continuously as its content change. Changes can happen according to either sliding window model, time or tuple-count based. Thus, we seek a dynamic and scalable organization of the contents of $\mathcal{B}$, able to report the results of ad-hoc top-k queries.

3.2 Arrangements

The *arrangement* $\mathcal{A}(\mathcal{S})$ of a finite collection $\mathcal{S}$ of geometric objects is the decomposition of the d-dimensional space into connected open cells of dimensions $0,\ldots,d$ induced by $\mathcal{S}$ [16]. Researchers have studied the arrangements of various geometric objects, including lines, curves, hyperplanes, hypersurfaces, triangles, circles, etc, for arbitrary dimensionality. Their applications span multiple scientific areas including Robotics and Computer Graphics just to name a few. Furthermore, many problems can be reduced to arrangement related equivalents. Detailed discussion is available elsewhere [20, 11, 3, 16].

3.2.1 Definitions and combinatorial complexity

An arrangement is comprised of cells of dimensionality ranging from 0 to d. A cell of dimensionality l, $0 \leq l \leq d$ is named *l-cell*. Cells of dimensionality 0, 1 and 2 are also called *vertices*, *edges* and *faces* respectively, while a cell of maximum dimensionality ($l = d$) is named *d-cell*. For example, Figure 2(a) depicts an arrangement of 3 lines ϵ_1, ϵ_2 and ϵ_3 in $\Re^2$. The regions of the plane denoted by $f_1,\ldots,f_7$ are the faces of the arrangement, $e_1,\ldots,e_9$ are its edges, while v_1, v_2, v_3 are its vertices.



(a) Example (b) The DCEL data structure

Figure 2: Arrangement of lines.

The *combinatorial complexity* of an arrangement is the overall number of cells of all dimensions in the arrangement. The combinatorial complexity of an l-cell is the number of cells of the arrangement of dimension less than l that are contained in the boundary of the cell. For example, the complexity of a face is the number of vertices and edges on its boundary. For arrangements of hypersurfaces, another interesting structure is the *zone* of a surface not present in the arrangement: it is the set of d-cells intersecting the surface. For example, in Figure 2(a), the zone of line ϵ_o is comprised of faces f_1, f_2, f_6. The complexity of a zone is the sum of complexities of the d-cells that comprise it.

In the case of arrangements of lines, the faces are convex, but can be unbounded (Figure 2(a), faces $f_1,\ldots,f_6$). Notice that the arrangement of Figure 2(a) consists of 3 vertices, 9 edges and 7 faces. It is possible to prove [16] that an arrangement of n lines is composed of $O(n^2)$ vertices, $O(n^2)$ edges and $O(n^2)$ faces. Therefore, the combinatorial complexity of an arrangement of lines is $O(n^2)$. Finally, the complexity of a single face is $O(n)$, since its boundary can be comprised of up to n edges.

One of the most important results in the arrangements literature is the Zone Theorem.

THEOREM 1. *The maximum complexity of the zone of a hyperplane in an arrangement of n hyperplanes in $\mathbb{R}^d$ is $\Theta(n^{d-1})$.*

For an arrangement of lines ($d = 2$), the complexity of a zone is $\Theta(n)$. This result is important, since for example, in two dimensions a line intersects n faces and the boundary of a face can have as many as n edges, therefore a trivial upper bound would be $O(n^2)$. Similar combinatorial complexity results also exist for arrangements of hyperplanes. Table 1 summarizes the complexity results for arrangements of n lines in $\mathbb{R}^2$ and n hyperplanes in $\mathbb{R}^d$.

Structure	Complexity	
	Lines	Hyperplanes
d-cell	$O(n)$	$O(n^{\lfloor \frac{d}{2} \rfloor})$
Zone	$\Theta(n)$	$\Theta(n^{d-1})$
Arrangement	$O(n^2)$	$O(n^d)$

Table 1: Summary of complexity results.

3.2.2 Representation

We briefly review efficient data structures for representing and storing arrangements and its various substructures; further details are available elsewhere [20, 11, 3, 16, 25]. The appropriate data structure for representing an arrangement depends on its intended use. For our purposes, we use the *doubly-connected-edge-list* (DCEL) data structure for arrangements of lines [25, 11]. This data structure has also been generalized for arrangements of hyperplanes.

The main idea behind the DCEL data structure is to represent each edge using a pair of directed *halfedges*, one going from the left to the right vertex of the edge and the other, known as its twin, going in the opposite direction. Beyond a simple and flat representation of the planar graph induced by the arrangement, the DCEL data structure maintains additional incidence and ordering information in order to facilitate its convenient and meaningful traversal. Figure 2(b) illustrates:

- Each halfedge maintains a pointer to its twin, e.g., e_2 to e_9 and vice versa.
- For each vertex, a circular list of the incident halfedges is maintained, in clockwise order. For example vertex v_3 maintains a list with edges (e_2, e_8, e_6, e_4).
- Each halfedge has pointers to its source and target vertices, for example edge e_1 to vertices v_1 and v_2.
- Each halfedge stores a pointer to its incident face, e.g., edges e_1, e_2, e_3 store a link to face f_1.
- For each face, the halfedges forming its boundary are organized in a doubly connected circular list. This way the boundary can be traversed in both clockwise and counterclockwise order. For example, edges e_1, e_2 and e_3 of face f_1 form a chain.

The total space complexity of the data structure is $O(n^2)$, the same as the combinatorial complexity of the arrangement. We will use the term arrangement to refer both to the partition of the plane and the DCEL data structure used to store it; we clarify in the specific context if needed.

4. INDEXING FOR TOP-K QUERY ANSWERING IN THE DUAL PLANE

Besides the primal plane, the top-k problem can be also mapped to the *dual plane*. The dual plane is a symmetric version of the primal plane where each point (line) in the primal plane is mapped to a line (point) in the dual. The mapping is not unique and can be selected so that it maintains certain geometric properties of interest to the problem at hand. In our case, each tuple $t = (x_1, x_2)$ is mapped to a line $\epsilon_t : y = (1 - x_2)x + (1 - x_1)$ in the dual plane. A query Q can be represented as a point $p(Q) = (\frac{w_2}{w_1}, 0)$, where w_1, w_2 are the weights of its scoring function. Then, the following Theorem holds.

THEOREM 2. *Consider n tuples $t_1, \ldots, t_n$, a scoring function $S = \vec{w}$ and the following mapping to the dual plane: $t_i \mapsto \epsilon_i : y = (1 - t_i.x_2)x + (1 - t_i.x_1)$ and $S \mapsto p(S) = (\frac{w_2}{w_1}, 0)$. Then, the ordering of $t_1, \ldots, t_n$ according to S is the same as the order in which a vertical ray originating from $p(S)$ and shooting upwards meets the corresponding lines ϵ_i (Figure 3).*

PROOF. Consider two tuples t, t' and the scoring function S, such that $S(t) < S(t')$. Let d_t ($d_{t'}$) be the vertical distance of line ϵ_t ($\epsilon_{t'}$) from point $p(S)$. We will demonstrate that $d_t > d_{t'} \Leftrightarrow S(t) < S(t')$. The value of d_t is equal to the y-coordinate of line ϵ_t for $x = \frac{w_2}{w_1}$. Therefore, $d_t > d_{t'} \Leftrightarrow (1 - x_2)\frac{w_2}{w_1} + (1 - x_1) > (1 - x'_2)\frac{w_2}{w_1} + (1 - x'_1) \Leftrightarrow w_2 - w_2 x_2 + w_1 - w_1 x_1 > w_2 - w_2 x'_2 + w_1 - w_1 x'_1 \Leftrightarrow (w_1 + w_2) - S(t) > (w_1 + w_2) - S(t') \Leftrightarrow S(t) < S(t')$. $\square$

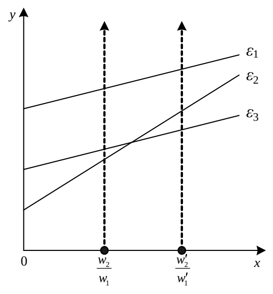
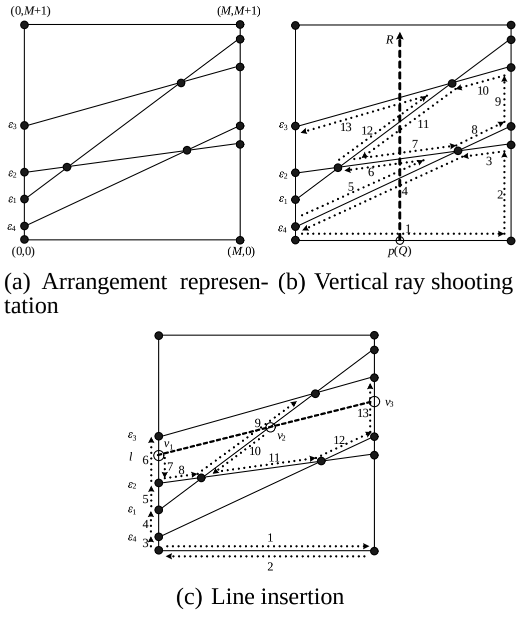


Figure 3: Top-k query answering in the dual plane.

This observation immediately points to an alternative solution to the top-k query answering problem. We can map the tuples of a data set to lines in the dual plane and then store and query the induced arrangement. The mapping to the dual plane and the use of arrangements provides an intuitive framework for representing and maintaining the rankings of *all* possible top-k queries in a non-redundant, self-organizing manner. The representation is non-redundant in the sense that all queries that produce the exact same tuple ranking are mapped to a continuous interval on the x axis and use the same part of the arrangement to retrieve that ranking. Notice that an intersection signifies a change in the ranking of two tuples. If we project all tuple intersections on the x-axis, all queries that lie between two consecutive projections will produce the same ranking. The representation is also self-organizing as the line insertion/deletion operations on the arrangement that we will subsequently describe, change appropriately the rankings of *all* queries, eliminating the need to identify and maintain the result of specific queries. Therefore, by mapping the top-k query answering problem to the dual plane, we obtain the capability to evaluate *ad-hoc* top-k queries by essentially storing and maintaining the rankings of all possible top-k queries in a non-redundant, query-independent manner.

4.1 Operating on the arrangement

Before we describe the algorithms for operating on the arrangement, let us make two observations. First, the points representing a query can only lie in the positive part of the x axis of the dual plane, so there is no need to maintain any arrangement related information (vertices, edges, faces) on its negative side. Second, since the domain of the tuples is the unit square, the lines that result after the mapping to the dual plane are of the form $y = ax + b$, where $0 \leq a, b \leq 1$. Therefore, for positive values of x, the lines lie exclusively on the positive quadrant. Notice that the selected mapping places all the elements of the top-k query answering problem in the positive quadrant of the dual plane. For this reason, we realize a bounded frame of dimensions $[0, M] \times [0, M + 1]$ (Figure 4(a)) in the dual plane and only store the part of the arrangement that lies inside the frame. However, all the combinatorial bounds that we described in Section 3.2.1 are still valid.



(a) Arrangement representation (b) Vertical ray shooting

(c) Line insertion

Figure 4: Representing and operating on an arrangement.

We need to determine a suitable value for M. Notice that we used a bounded frame in order to cover an unbounded quadrant. Care must be exercised so that all the relevant arrangement information is guaranteed to lie within the frame. Consider two lines $y = a_1 x + b_1$ and $y = a_2 x + b_2$. The x-coordinate of their intersection point is $x_I = \frac{b_2 - b_1}{a_1 - a_2}$. The maximum value of x_I is attained when $b_2 - b_1 = 1$ (the nominator is maximized) and $a_1 - a_2 = \delta$ (the denominator is minimized), where δ is the minimum allowable difference between two attribute values, as defined by either machine precision or attribute domain information. Therefore, we need to set $M > \frac{1}{\delta}$ in order to guarantee that all relevant information lies within the frame.

As we proved, a top-k query is mapped to a vertical ray shooting query in the dual plane. Figure 4(b) depicts the arrangement that corresponds to 4 tuples mapped to the dual plane (line segments $\epsilon_1, \ldots, \epsilon_4$ in our dual representation), the point $p(Q)$ corresponding to a query and the vertical ray R. An arrangement traversal starts from the bottom edge of the frame (step 1). This edge cannot intersect with any of the lines in the arrangement, so we use it as a reference point to initiate any traversal. The bottom edge is also associated with a face that lies inside the frame. Ray R moves inside that face until it intersects its boundary. We can walk along the boundary of the face until we locate the specific edge that intersects R (steps 2-4). In the configuration of Figure 4(b), the edge corresponds to tuple 4 (line segment e_4 in the dual representation), which is the highest ranking result for the query. At the intersection point, ray R leaves the current face and enters a neighboring face on the other side of the current edge. We therefore need to move to the twin of the current edge (step 5) and again move along the boundary of that neighboring face until we locate the new exit point of R (step 6). Since tuple 2 is associated with the edge at the exit point, it is the second tuple in the top-k result. This procedure is repeated (steps 7-11 and 12-13) until the desired number of results (all tuples in our example) has been retrieved. The pseudocode for this operation is presented in Algorithm 1.

Algorithm 1 Top-k retrieval algorithm

```
$R$: ray corresponding to query $Q$
$k$: number of tuples to be retrieved

Result=$\emptyset$
edge=bottom
edge=edge.next()

for $i = 1$ to $k$ do
    while not intersects($R$,edge) do
        edge=edge.next()
    end while

    Result.append(edge.tuple())
    edge=edge.twin()
    edge=edge.next()
end for

return Result
```

The insertion procedure is similar to the vertical ray shooting operation we just described. We use Figure 4(c) to demonstrate a line insertion. Every line in the arrangement intersects the left boundary of the frame. The first step is therefore to locate the edge along the left boundary where the new line intersects the existing arrangement. This involves a simple walk along the “exterior” of the frame. Notice that the rest of the plane outside the frame also constitutes an *unbounded* face, so we can also walk along its boundary (steps 1-6).

Let us denote by l the new line. After we locate the edge where l enters the arrangement, we need to split that edge at the intersection with l and insert a new vertex v_1 (step 7). Then, we traverse the boundary of the current face until we locate the edge where l exits (steps 8-9). This edge is also split at the intersection point with l and a new vertex v_2 is inserted. This and the last vertex inserted in the arrangement are connected using a new edge. This new edge corresponds to a segment of line l. Line l enters a new face (step 10) whose boundary we also need to traverse until we find the new exit point (11-13). The corresponding edge is again split (vertex v_3) and a new line segment is inserted in the arrangement as before. This procedure is repeated until the right boundary of the frame is reached. The pseudocode for this operation is presented in Algorithm 2.

The procedure for line removal is basically the reverse of a line insertion. The first segment of the line to be deleted is initially located along the left frame boundary and starting from there its segments are progressively removed. We therefore omit any further discussion on the delete procedure.

Let n be the number of lines that are already present in the arrangement. The left boundary of the frame is comprised of $n+1$ segments, as it is intersected by all n lines in the arrangement. Therefore, traversing the boundary until locating the proper segment where the line insertion will commence is an $O(n)$ operation. The complexity of the main loop of the insertion procedure is determined by the number of the arrangement's edges that must be traversed and the number of new edges that must be inserted. An edge insertion is an $O(1)$ operation. Since a line can intersect with up to n other lines, the cost for inserting the new edges is $O(n)$. As for the number of edges that must be traversed, their number is upper bounded by the total number of edges in the faces that the new line intersects. This is exactly the complexity of the line's zone and is of size $O(n)$ (Zone Theorem). Consequently, the total cost of

Algorithm 2 Insertion algorithm

```
$l$: line to be inserted

edge=bottom
edge=edge.twin()

while not intersects($l$,edge) do
    edge=edge.next()
end while

edge=edge.twin()
last_vertex=edge.split()

while edge not on right boundary do

    while not intersects($l$,edge) do
        edge=edge.next()
    end while

    edge.twin()
    current_vertex=edge.split()
    connect(current_vertex,last_vertex)
    last_vertex=current_vertex
end while
```

the insertion procedure is $O(n)$. In a very similar manner we can demonstrate that the cost of a deletion operation is also $O(n)$.

The cost of a query is determined by the number of the arrangement edges that must be traversed. Since a vertical ray can be treated as a line, the complexity of its zone is $O(n)$. This is an upper bound since a top-k query needs only to visit k faces, one face for each tuple that has to be retrieved. However, in the worst case, a face consists of n edges, therefore even though we walk along the boundary of k faces, the worst case cost of the procedure is still $O(n)$. Fortunately, we can expect the number of edge traversals that are required in practice to be limited, since the face size should normally be small.

To summarize, after mapping a data set of size n to the dual plane, we can treat it as an arrangement of lines and be able to perform ad-hoc top-k query answering, for any value of $k \leq n$, while also supporting tuple insertions and deletions in arbitrary order. The space complexity of the solution is $O(n^2)$ and the cost of the query answering, insert and delete operations is $O(n)$.

The aforementioned worst case bounds are a direct consequence of the combinatorial complexity of the arrangement, which might initially appear inappropriate for use in a streaming context. In order to compensate, we have developed a methodology that enables us to reduce to just a handful the number of tuples from the data set we need to store in the arrangement. The following section demonstrates our tuple pruning technique and derives more favorable complexity results for our core arrangement-based solution, namely $O(k \ln n)$ operations and $O(k^2 \ln^2 n)$ space consumption.

5. TUPLE PRUNING

Let Q be a top-k query. We denote by $R_k(Q)$ the point in the dual plane where a ray shooting upwards from $p(Q)$ (the point in the dual plane where Q is maps to) meets the k-th line. Furthermore, let $Q_1 < Q_2$ denote the fact that $p(Q_1)$ lies left of $p(Q_2)$ and let $[Q_1, Q_2]$ be the interval along the x axis of the dual plane between $p(Q_1)$ and $p(Q_2)$.

LEMMA 1. *Let Q_1, Q_2 be two top-k queries such that $Q_1 < Q_2$. Let also $l_1(Q_1, Q_2)$ be the line in the dual plane that passes through the origin and $R_k(Q_1)$. Then, any line that is located above $l_1(Q_1, Q_2)$ in the interval $[Q_1, Q_2]$ cannot be in the result* *of any top-k query that lies inside $[Q_1, Q_2]$.*

PROOF. The property of $l_1(Q_1, Q_2)$ is that it "bounds" the k lines in Q_1's result below it throughout interval $[Q_1, Q_2]$. Since all lines in the dual plane are of the form $y = ax+b$ with $0 \leq a, b \leq 1$, the "steepest" line that passes through $R_k(Q_1)$ is $l_1(Q_1, Q_2)$. The "steepest" lines that pass through $R_1(Q_1), \ldots, R_{k-1}(Q_1)$ are also bounded by $l_1(Q_1, Q_2)$. Therefore, the property holds. Any line ϵ that is located above $l_1(Q_1, Q_2)$ in the interval $[Q_1, Q_2]$ is also located above the k lines in Q_1's result throughout $[Q_1, Q_2]$. In other words, in the interval $[Q_1, Q_2]$ there are at least k tuples that are located below ϵ and hence score higher for any query that lies in $[Q_1, Q_2]$. Figure 5(a) illustrates for two top-2 queries. $\square$

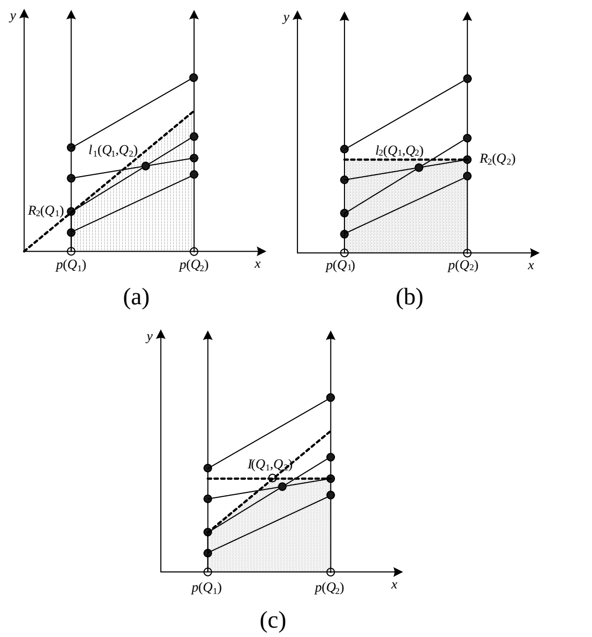
LEMMA 2. *Let Q_1, Q_2 be two top-k queries such that $Q_1 < Q_2$. Let also $l_2(Q_1, Q_2)$ be the* horizontal *line in the dual plane that passes through $R_k(Q_2)$. Then, any line that is located above $l_2(Q_1, Q_2)$ in the interval $[Q_1, Q_2]$ cannot be in the result of any top-k query that lies inside $[Q_1, Q_2]$.*

PROOF. The proof is similar to that of Lemma 2. Figure 5(b) illustrates for two top-2 queries. $\square$

We now introduce the following Theorem.

THEOREM 3. *Let Q_1, Q_2 be two top-k queries such that $Q_1 < Q_2$. Let also $I(Q_1, Q_2)$ be the intersection point of lines $l_1(Q_1, Q_2)$ (Lemma 1) and $l_2(Q_1, Q_2)$ (Lemma 2). We refer to this point as the* pruning point. *Then, any line that is located above $I(Q_1, Q_2)$ cannot be in the result of any top-k query that lies inside $[Q_1, Q_2]$.*

PROOF. Combining Lemmata 1 and 2, we can argue that any line that is located above either $l_1(Q_1, Q_2)$ or $l_2(Q_1, Q_2)$ in the interval between Q_1 and Q_2 cannot be in the result of any top-k query between Q_1 and Q_2. However, notice that this is true if and only if the line is above the intersection point of $l_1(Q_1, Q_2)$ and $l_2(Q_1, Q_2)$. Figure 5(c) illustrates for two top-2 queries. $\square$



(a) (b) (c)

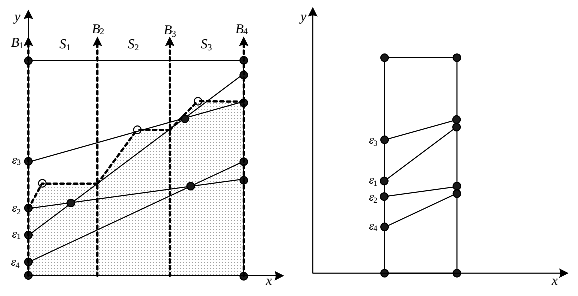
Figure 5: Proving Theorem 3.

Consequently, given two top-k queries Q_1, Q_2 and their result, we can, by performing simple computations, filter out a portion of the data set D that is definitely irrelevant to any top-k query in $[Q_1, Q_2]$. In other words, only the part of the data set not pruned, denoted by D^*, needs to be stored in the arrangement. This part contains all information relevant to *any* top-k query in $[Q_1, Q_2]$. Notice that D^* is not guaranteed to be minimal, in the sense that it can contain tuples that do not appear in the result of any top-k query in $[Q_1, Q_2]$ and thus are irrelevant. The guarantee we are offered is that $D - D^*$ contains only irrelevant tuples, while D^* contains all relevant tuples.

Pruning a significant portion of the data set involves a tradeoff. We can correctly answer any top-k query in $[Q_1, Q_2]$, requesting up to a number of results determined by the number of results (K) we choose to return for queries Q_1, Q_2. Notice that K determines the position of the pruning point. Effectively, the choice of K imposes a bound on the maximum number of top-k results any query can request ($k \leq K$) in accordance to previous work [24, 9].

Thus, we can utilize Theorem 3 in order to reduce the number of tuples we need to store in the arrangement and still retain the capability to answer any top-k query. Consider now a set B of $m+1$ top-k queries $B = \{B_1, \ldots, B_{m+1}\}$, such that $B_i < B_j$ for $i < j$ and $p(B_1) = (0, 0)$, $p(B_{m+1}) = (M, 0)$. We will refer to those queries as *borders*. In the dual plane, the rays corresponding to the borders superimpose on the arrangement a series of m disjoint, consecutive *strips* $S = \{S_1, \ldots, S_m\}$ that cover the entire arrangement. (Figure 6(a)). Treating each border as a query, we can compute (e.g., by traversing the arrangement along the vertical ray corresponding to the border) the query result for each border top-k query. As a result, we can compute the *pruning point* $I(S_i) = I(B_i, B_{i+1})$ for each strip and identify the part of the full data set D that we need to use in order to be able to answer any top-k query that lies inside a strip. We denote the filtered data set associated with strip S_i by D_i^*.

For example, in Figure 6(a), we have four borders $B_1, \ldots, B_4$ corresponding to top-3 queries, that induce strips S_1, S_2, S_3. The corresponding filtered sets are $D_1^* = \{\epsilon_1, \epsilon_2, \epsilon_4\}$ and $D_2^* = D_3^* = \{\epsilon_1, \epsilon_2, \epsilon_3, \epsilon_4\}$. Notice that there can be overlap between sets D_i^*



(a) Borders and strips (b) Arrangement strip

Figure 6: Placing borders on top of an arrangement.

There are two ways we could potentially utilize sets D_i^* in order to answer any arbitrary top-k query. The first is to consider the arrangement of data set $D^* = \bigcup_{i=1}^{m} D_i^*$. D^* contains all the tuples necessary to answer any top-k query, since it is a superset of any of the D_i^* data sets. We refer to this option as the *Full Arrangement* (FA) solution. A second option is to create a separate arrangement for each of the D_i^* tuple sets and use the relevant arrangement to answer a query. We will refer to each of these arrangements as *arrangement strips*. We call this option the *Strip Arrangement* (SA) solution.

The motivation behind the SA solution is that each of its arrangement strips can be considerably smaller than the full arrangement of the FA solution. Each of the arrangement strips indexes just a subset of the tuples indexed by the full arrangement. Furthermore, since each strip is responsible for answering queries that lie between two borders B_i and B_{i+1}, we only need to construct and maintain the arrangement in the interval $[B_i, B_{i+1}]$. This effect greatly reduces the arrangement complexity. Figure 6(b) depicts the arrangement that corresponds to strip S_2 of Figure 6(a). The complexity of the full arrangement is reduced in an arrangement strip in terms of points, segments and faces, while it is obvious that it can correctly answer any top-k query that falls inside the strip.

5.1 Handling updates

In a streaming environment we expect new tuples to enter the buffer $\mathcal{B}$ and old tuples to expire. As a result both the FA and SA solutions should support dynamic changes in the data they index. Let D denote the data set in our buffer at a time instance. Suppose that after a number of tuple insertions and deletions we end up with an updated version of D, denoted by D'. The first action one must take is to recalculate the result of the borders, in order to update the pruning points $I(S_i)$ associated with the strips. Using those updated points, we need to prune data set D' and calculate the filtered sets $D_1'^*, \ldots, D_m'^*, D'^*$. For the FA method, tuples in $D'^* - D^*$ must be added in the arrangement, while tuples in $D^* - D'^*$ must be removed. Similarly, for strip S_i of the SA method we must insert in the corresponding arrangement strip tuples in $D_i'^* - D_i^*$ and delete tuples in $D_i^* - D_i'^*$.

The two issues we have to resolve is how to update the borders and how to rapidly filter the data set D' in order to compute sets $D_1'^*, \ldots, D_m'^*, D'^*$. Let us first concentrate on efficiently filtering the data set. Given a set of n lines, we need to report the subset that lies below the pruning point. We would like to do so by considering as few of the n lines as possible. While we could potentially attempt to tackle the problem in the dual plane, it is much simpler and more convenient to handle it in the primal plane, as its equivalent (in the primal plane) is a well known Computational Geometry problem, namely *halfspace range searching*.

The halfspace range searching problem has the following form [2]: given a set of points in $\mathbb{R}^2$ ($\mathbb{R}^d$), we wish to index them using a data structure, so we can efficiently report all the points that lie above a query line (hyperplane). A connection between our filtering problem in the dual plane and the halfspace range searching problem in the primal plane is formally established by the following Theorem:

THEOREM 4. *Let t be a tuple and $p(t)$, ϵ_t its mapping to the primal and dual plane respectively. Then, line ϵ_t is located below a point $I = (x_I, y_I)$ iff point $p(t)$ is located above a line $\epsilon_I = f(I)$.*

PROOF. Let $t = (a, b)$. Then, $p(t) = (a, b)$ and $\epsilon_t = (1 - b)x + (1 - a)$. For ϵ_t to be below I, the following equation must hold: $y_I \geq (1 - b)x_I + (1 - a)$. For $x_I \neq 0$, this is equivalent to $b \geq -\frac{1}{x_I}a + (1 + \frac{x_I - y_I}{x_I})$, which is exactly the condition for $p(t)$ to be above line $y = -\frac{1}{x_I}x + (1 + \frac{x_I - y_I}{x_I})$. We get a similar result for $x_I = 0$. □

Theorem 4 allows us to perform filtering in the primal plane, where we can utilize specialized fully dynamic data structures to index the data in the buffer and perform halfspace range searching. Depending on query time and update time requirements, as well as potential space constraints, a large number of data structures could be employed, including grid or sophisticated partition tree structures, that offer $O(n^{\lfloor d/2 \rfloor - 1 + \epsilon})$ update time and $O(\log n + r)$ query time [2], where d is the dimensionality, n the number of points indexed and r the result size of the query.

In order to dynamically calculate and maintain the top-k query result for the borders, we observe that this is exactly the problem of maintaining top-k query results for a fixed set of queries in a streaming environment. For this problem we can utilize known solutions, such as [26, 19]. We discuss our choices in Section 8.1.

5.2 Pruning efficiency

As was previously discussed, the filtered data set D^* contains all tuples that can potentially appear in the result of a top-k query, a property that guarantees the correctness of the proposed method. Because of this property, D^* is a superset of the *k-skyband* [19] of D. The k-skyband is the generalization of the *skyline* of a tuple set. As the skyline is the minimal subset of D required to answer correctly any top-1 query, the k-skyband is the minimal subset required to correctly answer any top-k query.

Therefore, the size of D^* is lower bounded by the size of the k-skyband. As will become clear in the following section, increasing the number of borders results in more efficient pruning. Actually, a sufficient number of borders can reduce D^* to the k-skyband. However, this is not required since, as we will demonstrate in Section 8, a small number of borders can provide a sufficiently tight superset D^*.

This observation allows us to approximate the size of D^* with the size of the k-skyband. In [8] it is established that for n uncorrelated d-dimensional tuples, the size of the skyline is $\Theta(\frac{\ln^{d-1} n}{(d-1)!})$. While this result has not been generalized for the k-skyband, it is plausible to estimate its size as $\Theta(k\frac{\ln^{d-1} n}{(d-1)!})$. For $d = 2$, this is equal to $\Theta(k \ln n)$. Furthermore, in [15] it is demonstrated that even for anticorrelated data and high dimensionality, the size of the skyline does not explode and remains a tiny fraction of the original data set.

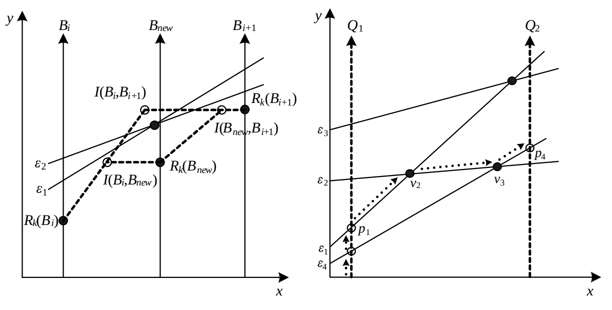
Putting it all together, after the application of our pruning technique, only $|D^*|$ tuples need to be stored in an arrangement representation. Therefore, the complexity of arrangement operations is reduced to $O(|D^*|)$ instead of $O(n)$, n being the size of the buffer. In the case of uncorrelated data, the cost of arrangement operations is only $O(k \ln n)$.

6. PLACING THE BORDERS

We now turn our attention to the problem of placing a number of borders on the arrangement and discuss the issues associated with the dynamic maintenance of such placement under changing data distributions.

Let us first focus on the tradeoffs associated with the number of borders placed on the arrangement. A first observation is that increasing their number results in more efficient pruning. Figure 7(a) illustrates this effect. A new border B_{new} is added in between two existing borders B_i and B_{i+1}. As a consequence, the region that can be pruned given the newly created pruning point is larger than before. In the worst case, it will be exactly as before the addition of the new border, since $R_k(B_i) \leq R_k(B_{new}) \leq R_k(B_{i+1})$. For example, in Figure 7(a) notice that lines ϵ_1 and ϵ_2 are pruned only after the addition of the new border.

Although increasing the number of borders increases the pruning capability, so does the cost one has to pay after each update in order to maintain the correct top-k values at each border and correctly assess the pruning point for each strip. Furthermore, in the case of the SA method, so does the number of arrangement strips that need to be maintained. As the number of borders increases, we expect to get diminishing increases in pruning efficiency, while their maintenance becomes more costly. We explore such tradeoffs in Section 8.



(a) Insertion of a new border (b) Evaluation of band-queries

Figure 7: New borders and band-queries.

We will base the description of our border positioning strategy on the SA technique. The method can be directly used to generate an appropriate border set for the FA technique. The borders induce a partitioning of the dual plane into strips and the complexity of the corresponding arrangement strips determines the performance of queries and update operations on them. Assuming a uniform query workload, a natural objective is to produce strips that are equally complex. In general, the number of vertices in a strip provides a very accurate estimate of its complexity, that is also easy to compute without having to actually materialize the strip. The intuition behind the strategy is simple: a single dense and complex strip can dominate update and query time; thus by equalizing strip complexity, given a specific number of strips, we aim to provide an unbiased treatment to all queries and update events. An additional plausible objective would be to minimize the maximum complexity of a strip, that is minimize the maximum number of vertices within a strip, for a specific number of strips.

Given a set of points D in the buffer, placing a number of borders B in a way that we generate arrangement strips of equal complexity is equivalent to the problem of generating an *equi-depth* partitioning of the arrangement into B arrangement strips. This problem is challenging as even the placement of a single border will create two new strips of unknown size. The specific size depends on how data are distributed inside the buffer. We don't have a way of estimating the size unless we actually place the border and generate the two arrangement strips. One way to circumvent this issue would be to assume that we have information regarding the sizes of resulting arrangement strips, for a large number of strips $B' \gg B$. Thus, we first generate a number B' of arrangement strips, placing them in an *equi-width* manner, while maintaining the number of vertices in each.

Although this would reduce the problem to a one dimensional problem, an additional complication arises. Let $H_w(B')$ denote the resulting array recording the number of vertices in each of the B' strips. We can then solve the equi-depth partitioning problem assuming a "granularity" of B' strips and induce B strips in an equi-depth manner (or according to some other objective, say minimize L_{inf} norm), applying known algorithms [21]. However, every time we collapse a number of adjacent strips, the sum of their sizes is only a crude *lower bound* of the size of the newly formed strip. For example, in Figure 7(a), after merging strips $[B_i, B_{new}]$ and $[B_{new}, B_{i+1}]$ the size of the resulting strip is equal to the sum of the sizes of the merged strips plus the vertex that is introduced by the intersection of lines ϵ_1 and ϵ_2 that are no longer pruned. In practice, this effect is greatly exacerbated. As a result, any attempt to obtain a partitioning based on maintaining information at a lower granularity will be always based on lower bound estimates. Such lower bounds, do not necessarily provide a good indication of the actual size (complexity) of arrangement strips.

We demonstrate that even aiming to maintain an equi-depth partitioning using such lower bounds is a hard problem. Let $H_d(B)$ denote the resulting equi-depth histogram (on the lower bounds of the complexity of the arrangement strips), with the bucket boundaries signifying the location of the B borders. This will generate an equi-depth partitioning into arrangement strips, for the initial contents of the buffer before any update commences. A single update, in terms of points entering or leaving the buffer can affect all entries of $H_w(B')$ in the worst case. In principle, all the B' equi-width strips may be affected. However, the values in $H_w(B')$ can be incrementally maintained as updates take place in the buffer.

We would like to dynamically maintain the boundaries of $H_d(B)$ as changes happen into $H_w(B')$. This is a challenging streaming problem, as the model of streaming changes into $H_w(B')$ does not follow the usual sliding window model. In particular the model that describes the streaming changes into $H_w(B')$ is the cash-register model [14]. In this model, the only results known maintain a sketch of the changes in $H_w(B')$ and periodically extract a V_{opt} (minimizing the L_2 norm) [18] histogram from the sketch.

Thus, the optimization is not continuous but happens periodically at select time intervals. No results are known for equi-depth histograms or other norms (e.g., the L_{inf}) in this model. Even if one is interested to pursue extraction of V_{opt} histograms, given that the changes to the underlying $H_w(B')$ can be arbitrary under buffer updates, the resulting V_{opt} boundaries (locations of the borders) can be vastly different than the ones before the updates. In the worst case adjusting the location of the borders may be worst than building the new arrangement strips from scratch. This is because the overhead of identifying the relevant vertices in the arrangement strips between the old and new border positions for all borders would result to very high maintenance and pruning overhead, since changes are not localized.

For these reasons, we chose to adopt a conservative approach to this problem and rebuild the arrangement strips when necessary. The cost of doing so is amortized over time. As a result, our overall strategy for maintaining the locations of the borders consists of:

- Initially placing them in a equi-depth fashion. We do so by first considering the arrangement on the entire buffer and identifying its vertices. We then sweep the arrangement plane until we encounter half its vertices and place a new border at this point. We empirically found that this border positioning protocol creates two strips of approximately equal complexity. Then, we recursively split the resulting strips, until each arrangement strip contains at most a specified number of vertices v_{max}. We experimentally study the different effects of varying the value of v_{max} in Section 8.

- Allowing changes to occur in the buffer and conducting the suitable arrangement maintenance to assure correctness (inserting and deleting tuples from the affected arrangement and arrangement strips).

- Monitoring appropriate statistics to validate that the goals of the border generation strategy we just described are met. For the SA method, two statistics are monitored: the mean and coefficient of variation (standard deviation over the mean) of the arrangement strips' complexity (as is measured in terms of arrangement vertices). The goal of our border generation method is to produce strips of approximately equal complexity that also have fewer than v_{max} vertices. Therefore, the coefficient of variation is monitored to guarantee that strips are of similar complexity, while the mean is monitored to assert that our maximum strip complexity objective is not violated. For the FA method, we monitor the arrangement's complexity. The number of vertices after the last rebuild is saved, and if the current number of vertices becomes considerably larger than the initial value, a significant change in the data distribution is implied.

- Triggering a full rebuild when the aforementioned statistics exceed predefined thresholds. The rebuild involves the generation of new border set and the reconstruction of the induced arrangement or arrangement strips. We evaluate this strategy in Section 8.

7. EXTENSIONS

7.1 Band-queries

Consider the following interesting type of query. Given a continuous band of queries $[Q_1, Q_2]$ (Section 5), return all the tuples that appear in the top-k result of *any* query in $[Q_1, Q_2]$ (union semantics) or return the tuples that appear in the top-k result of *all* queries in $[Q_1, Q_2]$ (intersection semantics). Band queries can be relevant and useful in many contexts. As an example, a user might not wish to provide specific query weights, but specify instead weight *ranges*. Such a query can be translated into a band query in a straightforward manner.

Figure 7(b) helps demonstrate how band queries can be easily evaluated using the proposed techniques. Suppose that we want to answer a top-2 band query between Q_1 and Q_2. We first perform a vertical ray shooting query at Q_1 and retrieve its top-2 result $\{\epsilon_4, \epsilon_1\}$. The segment of ϵ_1 that intersects Q_1 is used as a starting point to further traverse the arrangement towards Q_2. The invariant we maintain while progressing towards Q_2 is that we always move along the top-2 (top-k in general) tuple from the base of the arrangement. The vertices we meet during the traversal can correspond to tuples leaving and entering the "current" top-2 result (vertex v_2) or just a reordering between the top-1 and top-2 tuples (top-k and top-$(k-1)$ in general). Depending on whether we use union or intersection semantics, we keep adding or removing tuples from the initial top-2 set, until we reach Q_2.

The part of the arrangement that needs to be traversed in order to answer a band query is known as its k-level, and its combinatorial complexity is $O(n\sqrt[3]{k})$ [3, 16]. Therefore, band queries can be efficiently evaluated in $O(|D^*|\sqrt[3]{k})$ time.

7.2 Higher dimensionality

A direct extension of our techniques to d-dimensions will require us to compute the arrangement of $|D^*|$ $(d-1)$-dimensional hyperplanes, each corresponding to the dual of an original point in the buffer. As we presented in Section 3.2.1, the complexity of such an arrangement is $O(|D^*|^d)$. Moreover, to answer any query, we will have to efficiently trace a ray (the dual of a query) within this arrangement and compute the first k intersections with the hyperplanes. The complexity of this operation is bounded by the complexity of the zone of a 1-dimensional line in this arrangement. The extended Zone Theorem in [4] states that this complexity is $O(|D^*|^{(d+1)/2})$. Furthermore, in Section 5.2 we observed that the size of $|D^*|$ in the case of uncorrelated data is $\Theta(k\frac{\ln^{d-1} n}{(d-1)!})$. As a consequence, the direct extension of our approach to three dimensions will result in an $O(k^3 \ln^6 n)$ arrangement of hyperplanes and the time complexity of answering any query will be $O(k^2 \ln^4 n)$.

As an alternative, we can also leverage the efficient solution involving arrangements we presented for two dimensions in the following approach. Consider a d-dimensional space with dimensions $x_1, x_2, \ldots, x_d$. Let us form pairs of dimensions, such as (x_1, x_2), $(x_3, x_4), \ldots, (x_{d-1}, x_d)$ (wlog, assume d is even). We can project the points in the buffer into each of these two dimensional spaces, and build arrangements for the duals of each. Now, given a query such as $w_1x_1 + w_2x_2 + \cdots + w_dx_d$, we can similarly partition it into $\frac{d}{2}$ 2-dimensional queries: $w_1x_1 + w_2x_2$, $w_3x_3 + w_4x_4$ and so on. Each partitioned query can be answered by its corresponding arrangement using the techniques we have developed thus far. We can then use the Threshold Algorithm (TA) to collect the sorted order of the tuples in each projection, and merge them to obtain the top-k points of the original query.

8. EXPERIMENTAL EVALUATION

In this section we discuss in greater detail the implementation of the proposed methods and present a thorough experimental study evaluating their performance.

8.1 Implementation details

In Section 5.1 we introduced a framework for updating the proposed indexing structures that requires concrete methods for performing halfspace range searching and border maintenance. Since a border is just a persistent top-k query, we use the query monitoring technique of Mouratidis et al. [19] to perform border maintenance. The technique uses a simple, regular grid to index the buffer. Since the grid is in place, we also utilize it to perform halfspace range searching. Although this choice does not offer optimal halfspace query answering performance, it relieves the proposed methods from the overhead of maintaining a second, dedicated data structure. Even better performance can be achieved by the techniques proposed herein by utilizing optimum structures for halfspace range searching [2]. Our implementation of the arrangement data structure was based on the *Arrangements* package of the CGAL library [25] and supports lines in $\mathbb{R}^2$.

8.2 Experimental setting

A stream of tuples arrives continuously at the system which maintains a tuple-count buffer of size n. Varying tuple arrival rate is "simulated" as follows: at each *update cycle*, $r\%$ of the tuples in the buffer are evicted to be replaced by an equal number of fresh, incoming tuples. After each buffer update, all indices are brought up to date in order to reflect the current state of the buffer and m ad-hoc top-k queries are evaluated.

The parameters that can potentially affect performance are the size of the buffer (n), the rate of the incoming stream (r), the number of queries monitored (m) and the number of results retrieved by each query (k). Attribute distributions of the incoming tuples can also have a direct impact on performance. Therefore, we designed a set of experiments to evaluate the actual impact of these parameters to the performance of our methods, FA and SA.

We also compare the performance of our methods with the top-k answering technique of [19] (Section 2), referred to as MBP. All experiments were carried out on a 3.2GHz Pentium D CPU with 2GB of RAM.

8.3 Experimental evaluation

8.3.1 *Pruning efficiency*

The tunable parameter of the MBP method is the grid granularity. In [19], the use of a specific, empirical value, is suggested but it is intended for use with a restricted set of buffer size values. We found the best performing value to be dependant on the buffer size, therefore for every different value of n in our experiments we used the respective best performing grid granularity for the MBP technique. As a heuristic and for fairness, we also used the same granularity for the relevant grid utilized by the FA and SA methods.

Respectively, the tunable parameter for the FA and SA solutions is the maximum allowable strip complexity v_{max} (Section 6). As Figure 8(a) demonstrates, the number of borders generated is inversely proportional to v_{max}. This is expected, as a larger number of borders is required to produce lower complexity arrangement strips. Figure 8(a) also depicts the complexity of the resulting full arrangement (for the FA solution) and the total complexity of the induced arrangement strips (for the SA solution). As expected, larger values of v_{max} result in fewer borders, therefore worse tuple pruning and more complex arrangements.

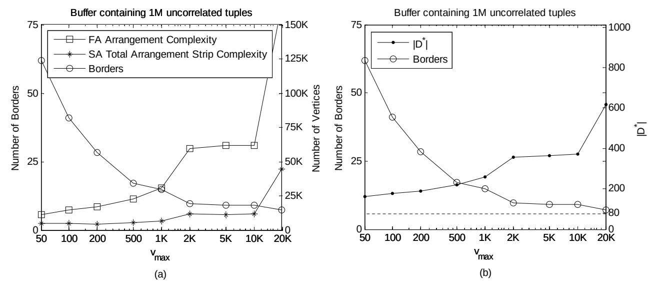


Figure 8: Effect of v_{max}.

Figure 8(b) demonstrates the effectiveness of our pruning technique in terms of the size of the filtered dataset D^*. Even a small number of borders, i.e., 16 borders for $v_{max} = 1K$, produces a filtered dataset D^* containing on average only 250 tuples, a tiny fraction of the 1M tuples that are present in the buffer. The size of the k-skyband ($k = 20$ for this experiment), the minimum size for D^* we could hope to achieve is approximately 80 tuples.

Summarizing, larger values of v_{max} result in fewer borders, but larger arrangement (strips) for the FA (SA) method. Therefore, less time is spent in border maintenance and tuple filtering (remember that there is a filtering step per strip), while more time is spent in arrangement maintenance and query answering. Smaller values of v_{max} have the exact opposite effect. Therefore, the best value of v_{max} is subject to such bicriteria reasoning; a reasonable approach is to balance these competing trends. We experimentally identified that the value of v_{max} depends primarily on the size of the streaming buffer and is insensitive to the rest of the experimental parameters, including data distribution. Table 2 summarizes the best value of v_{max} identified for various values of n (in terms of combined ad-hoc query and update performance).

Buffer Size	v_{max}	
	FA	SA
10K,20K,50K,100K	100	500
200K,500K	200	1000
1M	500	1000
2M	500	5000
5M,10M	1000	5000

Table 2: Best performing v_{max} values.

It is worth noticing in Figure 8(a) that for the same number of borders (and border positioning), the total complexity of the arrangement strips of the SA solution is always less than the corresponding arrangement complexity of the FA solution. The reason is the following: the arrangement strips of the SA method isolate the interaction between lines only within relevant strips. Consider for example two lines ϵ_1 and ϵ_2 that are materialized within an arrangement strip S. If their intersection point lies outside S, it is not stored in the arrangement strip. However, if these two lines are materialized in the full arrangement utilized by the FA method, their intersection point and all the relevant induced information will have to be stored, resulting in additional complexity. Furthermore, consider a line ϵ_1 that is stored only in arrangement strip S_1 and a line ϵ_2 that is only stored in arrangement strip S_2. Possible interaction between these two lines is irrelevant for the SA solution, but not for the FA method.

It should also be noted that although the proposed techniques utilize arrangements data structures that are of quadratic space complexity, the resulting storage overhead attributed to the arrangements is negligible. Figure 9 demonstrates.

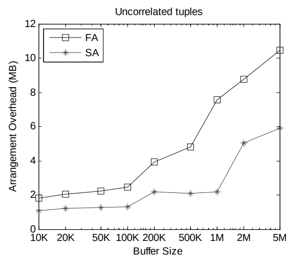


Figure 9: Arrangements space consumption.

8.3.2 *Evaluating the performance of the techniques*

In our first set of experiments, we used randomly generated 2-dimensional data and queries to evaluate the effect of each parameter n, r, m and k independently. The default values for the parameters is $n = 1M$, $r = 1\%$, $m = 1K$ and $k = 20$. In each experiment, a single parameter was varied, spanning diverse values, while the rest of the parameters were set to their default values. For each parameter set, three different incoming tuple distributions were used: uniform (tuple attributes x_1 and x_2 are uncorrelated), correlated (high values of x_1 imply high values of x_2) and anticorrelated (high values of x_1 imply low values of x_2). The data were generated as suggested in [7]. For each parameter set and data distribution we populated the buffer and then measured the total time required for $c = 100$ update cycles. No rebuilds were required by the proposed methods in this experiment, since only static data distributions were used.

Figures 10-13 present the results of this experiment set. The plots for correlated data are omitted as the performance of all three methods was found on all instances to be very similar to their corresponding performance on uniform data. We have used a logarithmic Time axis to present the experimental results, as the performance of the proposed solutions is frequently orders of magnitude better than the competing MBP technique.

In general, we found the proposed solutions to consistently outperform the MBP technique by a large margin, even by two orders of magnitude in the case of anticorrelated data, on which MBP performs particularly bad. On the contrary, the border positioning scheme employed by both the SA and FA solutions adapts extremely well to all data distributions, guaranteeing predictable performance while not needing a different, appropriate v_{max} value for every different data distribution. Furthermore, notice that the SA method is consistently better than the FA method throughout the parameter space and for all three data distributions. The reason is the reduced total complexity of the arrangement strips of the SA method with respect to the full arrangement complexity of the FA method.

More specifically, Figure 10 shows that the FA and SA solutions scale gracefully with respect to the buffer size, consistently outperforming MBP for buffer sizes as small as 10K to as large as 5M,

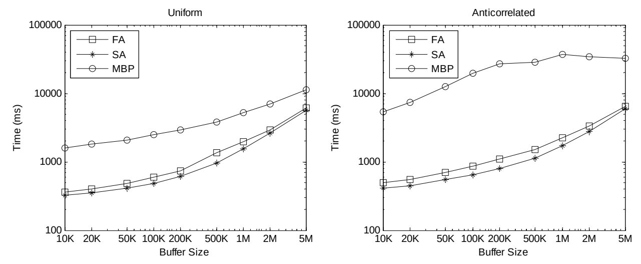


Figure 10: Time vs Buffer Size.

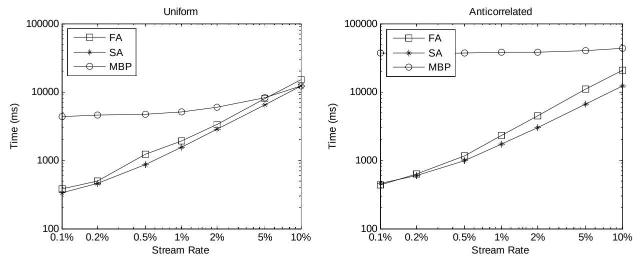


Figure 11: Time vs Stream Rate.

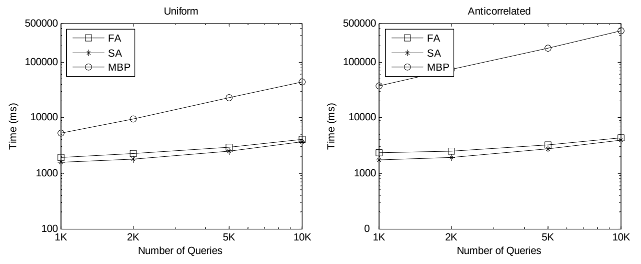


Figure 12: Time vs Number of Queries.

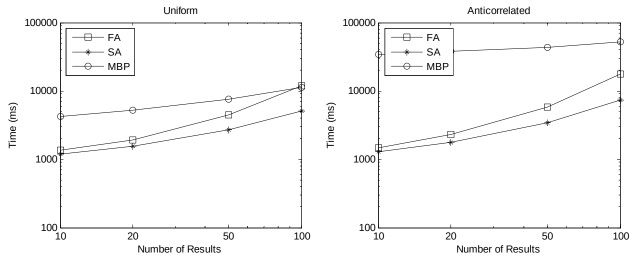


Figure 13: Time vs Number of Results (k).

and more. Figure 11 demonstrates that the proposed methods scale linearly with respect to the incoming stream rate and offer a clear advantage for lower stream rates. As the stream rate increases, this performance advantage diminishes, due to the fact that the MBP method has an insignificant index maintenance overhead (a simple regular grid).

Both the FA and SA techniques offer rapid query evaluation that is also decoupled from index maintenance. Therefore, increasing the number of queries has a small impact on their total runtime. On the contrary, MBP scales particularly bad with respect to the number of queries. We can witness this trend in Figure 12. Finally, Figure 13 demonstrates that the SA method is more scalable than FA with respect to k and as scalable as MBP. The performance advantage of SA over FA in this case can again be attributed to the interaction isolation effect, that becomes more pronounced as the value of k increases.

The performance of the proposed solutions was also evaluated using real data. To this end, we utilized the Intel Lab Sensor Data [1]. The data set contains a stream of tuples consisting of temperature, humidity, luminosity and voltage readings, collected periodically by 54 sensors. We processed this data set, extracting a 2-dimensional stream of total size 11M tuples. The data distribution of the resulting stream is both irregular and, more importantly, changing over time. Therefore, this experiment evaluates the robustness of the proposed methods when the incoming data distribution demonstrates high variability. Both the SA and FA solutions avert potential performance degradation by rebuilding their indices when the data distribution changes, amortizing the cost incurred by rebuilding over time.

For this experiment we used the default values for r, m and k and varied the value of n. The total time required for $c = 1000$ update cycles was measured. For a buffer size of $1M$ tuples, we process the entire stream of 11M tuples. For smaller values of the buffer size, we ignore an appropriate number of incoming tuples before actually processing a tuple, so that all 11M tuples of the stream arrive at the system in only 1000 cycles. For example, if $n = 200K$ we process 1 in every 5 tuples and if $n = 10K$ we process 1 in every 100 tuples. This way we witness the same pattern of variation in the stream for all buffer size values. Lastly, we used the following rebuild protocol (Section 6). The SA method triggered a rebuild when either the mean value of arrangement strip complexity exceeded $1.5v_{max}$, or when the corresponding coefficient of variation exceeded value 2.5. The FA method method triggered a rebuild whenever it witnessed a 3-fold increase in arrangement complexity since the last rebuild. The aforementioned threshold values were also validated on a variety of synthetic, changing distributions and were found to yield near optimal performance on all occasions.

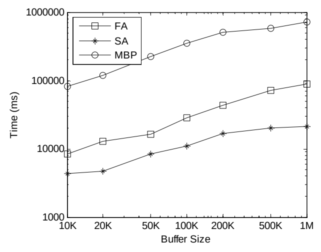


Figure 14: Time vs Buffer Size, using real data.

The results of the experiment are presented in Figure 14. The SA solution is considerably more robust from both the MBP and FA techniques. The FA method performs worse than SA particulary due to a higher number of rebuilds required. This effect is exacerbated for large buffer size values, where the overhead incurred by a rebuild is larger. The reduced number of rebuilds required by the SA method is once more due to the methods' ability to isolate changes within few strips: updates that negatively affect few arrangement strips in the SA method, have a global adverse effect on the full arrangement of the FA method. While the SA method can tolerate a few oversized strips, without considerable performance degradation, the FA method cannot maintain an oversized full arrangement without a severe impact on its performance.

We also performed experiments implementing the query partitioning technique described in Section 7.2, based on the 2-dimensional SA solution. Figure 15(a) presents the results of the experiment, involving 3-dimensional uniformly distributed tuples and tuples from the real data set. Furthermore, k was set to 5 and we used the same rebuild threshold values as before. For higher dimensions the technique appears more sensitive to the size of the buffer, since due to the multiple arrangements realized, the number of points involved in query answering increases. In the figure we report the time required to report results which includes the time to compute the answers using the TA algorithm, as the buffer size increases.

Notice that in general, and especially in higher dimensions, the performance of the MBP technique is expected to be highly sensitive to the choice of the grid granularity. In the 2d experiments presented before, the grid granularity for the MBP technique, was unrealistically set to the empirically observed value that yielded best performance for each specific data distribution. That way, the MBP technique was maximally favored. One major deficiency of the MBP technique is that it is difficult to set the grid granularity since it depends on a multitude of factors.

Figure 15(b) presents the performance of the MBP technique with respect to the underlying grid granularity. Notice that the performance of MBP is (a) extremely sensitive to the grid size and (b) the best performing grid size strongly depends on the data distribution. Furthermore, the best performing grid granularity on uniform data, when used on real data results in particularly poor performance and vice versa. In [19] no proposal was presented on how to choose or adjust the grid size. It is evident that the right choice of grid granularity for good performance depends on the data distribution. A similar trend was observed for all buffer size values. The figure shows that both for uniform and real data sets (similar results observed for correlated and anti-correlated data) SA offers very large performance benefits for a vast range of grid sizes.

We have also performed higher dimensional experiments involving larger values of k, making similar observations. We omit these results due to space constraints.

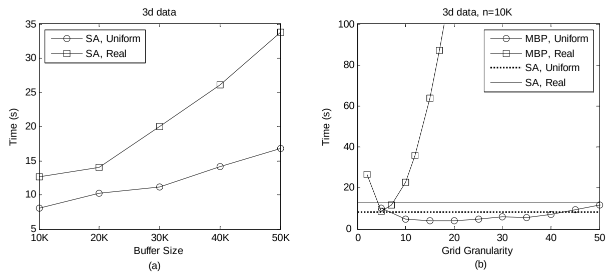


Figure 15: Higher dimensionality experiments.

9. CONCLUSION

In this paper we have presented techniques for answering ad-hoc top-k queries over streaming data. We have introduced techniques based on the notion of geometric arrangements and presented their practical realization in a data streaming scenario. We have presented analytical and experimental results quantifying the tradeoffs around the choice of various parameters inherent in our techniques. Our results demonstrate the practical utility of our methods.

10. ACKNOWLEDGEMENTS

The work of Dimitrios Gunopulos was supported by NSF (IIS 0330481, IIS 0534781). The work of Gautam Das was partially supported by grants from Microsoft Research and startup funds from the University of Texas at Arlington.

11. REFERENCES

[1] Intel lab data, 2004. http://db.csail.mit.edu/labdata/labdata.html.

[2] P. Agarwal and J. Erickson. Geometric Range Searching and Its Relatives. In *Advances in Discrete and Computational Geometry, Contemporary Mathematics 223*.

[3] P. K. Agarwal and M. Sharir. Arrangements and Their Applications. In *Handbook of Computational Geometry*, chapter 2, pages 49–119. Elsevier, 2000.

[4] B. Aronov, M. Pellegrini, and M. Sharir. On the Zone of a Surface in a Hyperplane Arrangement. *Discrete Comput. Geom.*, 9(2):177–186, 1993.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *PODS*, pages 1–16, 2002.

[6] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. IO-Top-k: Index-access Optimized Top-k Query Processing. In *VLDB*, pages 475–486, 2006.

[7] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, pages 421–430, 2001.

[8] C. Buchta. On the Average Number of Maxima in a Set of Vectors. *Inf. Process. Lett.*, 33(2):63–65, 1989.

[9] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The Onion Technique: Indexing for Linear Optimization Queries. In *SIGMOD*, pages 391–402, 2000.

[10] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering Top-k Queries Using Views. In *VLDB*, pages 451–462, 2006.

[11] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications (2nd Edition)*. Springer-Verlag, 2000.

[12] R. Fagin. Combining Fuzzy Information from Multiple Systems. *PODS*, pages 216–226, June 1996.

[13] R. Fagin. Fuzzy Queries In Multimedia Database Systems. *PODS*, pages 1–10, June 1998.

[14] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. Fast, Small-space Algorithms for Approximate Histogram Maintenance. In *STOC*, pages 389–398, 2002.

[15] P. Godfrey. Skyline Cardinality for Relational Processing. In *FoIKS*, pages 78–97, 2004.

[16] D. Halperin. Arrangements. In *Handbook of Discrete and Computational Geometry*, chapter 24, pages 529–562. CRC Press, 2004.

[17] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. In *SIGMOD*, pages 259–270, 2001.

[18] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal Histograms with Quality Guarantees. *VLDB*, pages 275–286, 1998.

[19] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous Monitoring of Top-k Queries over Sliding Windows. In *SIGMOD*, pages 635–646, 2006.

[20] J. O'Rourke. *Computational Geometry in C (2nd Edition)*. Cambridge University Press, 1998.

[21] G. Piatetsky-Shapiro and C. Connell. Accurate Estimation of the Number of Tuples Satisfying a Condition. In *SIGMOD*, pages 256–276, 1984.

[22] R. Fagin and A. Lotem and M. Naor. Optimal Aggregation Algorithms For Middleware. *PODS*, June 2001.

[23] M. Theobald, G. Weikum, and R. Schenkel. Top-k Query Evaluation with Probabilistic Guarantees. In *VLDB*, pages 648–659, 2004.

[24] P. Tsaparas, T. Palpanas, N. Koudas, and D. Srivastava. Ranked Join Indicies. *IEEE ICDE*, Mar. 2003.

[25] R. Wein, E. Fogel, B. Zukerman, and D. Halperin. 2d Arrangements. In *CGAL-3.2 User Manual*. 2006.

[26] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient Maintenance of Materialized Top-k Views. In *ICDE*, pages 189–200, 2003.