Dynamic Processing of Dominating Queries with Performance Guarantees

A. Kosmatopoulos
Dep. of Informatics
Aristotle University
Thessaloniki, Greece
akosmato@csd.auth.gr

K. Tsichlas
Dep. of Informatics
Aristotle University
Thessaloniki, Greece
tsichlas@csd.auth.gr

A. N. Papadopoulos
Dep. of Informatics
Aristotle University
Thessaloniki, Greece
papadopo@csd.auth.gr

ABSTRACT

The top-k dominating query returns the k database objects with the highest score with respect to their dominance score. The dominance score of an object p is simply the number of objects dominated by p, based on minimization or maximization preferences on the attribute values. Each object (tuple) is represented as a point in a multidimensional space, and therefore, the number of attributes equals the number of dimensions. The top-k dominating query combines the dominance concept of skyline queries with the ranking function of top-k queries and can be used as an important tool in multi-criteria decision making systems. In this work, we focus on the 2-dimensional space and present, for the first time, novel algorithms for top-k dominating query processing in main memory with non-trivial asymptotic guarantees. In particular, we focus on both the semi-dynamic case (only insertions are allowed) and the fully-dynamic case (insertions and deletions are supported). We perform a detailed cost analysis regarding the worst-case complexity of preprocessing, the worst-case complexity for the query cost and the amortized complexity for updates (insertions and deletions) focusing on the RAM computation model. Our solutions require space linear with the number of points, which is very important especially for modern applications which manipulate massive datasets. In addition, we discuss the case of the word-RAM computation model, where slightly better results are obtained.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*; H.2.2 [**Database Management**]: Physical Design —*Access methods*; E.1 [**Data**]: Data Structures

General Terms

Theory, Algorithms, Performance

Keywords

dynamic queries, top-k domination

1. INTRODUCTION

Recently, there has been an increasing interest in *preference-based queries*, due to their ability to select the most interesting objects of a given dataset. The data objects are characterized by a number of usually contradictory attributes, such as price and quality, and therefore, selecting a suitable result becomes a challenging task.

As an example, consider a hotel database, where each hotel is represented as a 2-dimensional point with two attributes (dimensions): (a) its distance from the conference venue and (b) the price for a standard room per night. Generally, a potential customer would be interested in hotels that have both of these attributes as small as possible. The solution would then consist of all hotels that are in a sense more "preferred" than others. In the sequel, each object will be represented as a point in a multidimensional space, where each dimension corresponds to an attribute. The dataset of points is denoted as $\mathcal{S}$. Our work is based in the concept of *domination* (or dominance) which is defined as follows:

DEFINITION 1 (DOMINATION). *A point $p \in \mathcal{S}$ dominates another point $q \in \mathcal{S}$, and we write $p \prec q$, iff p is as good as q in all dimensions and it is strictly better than q in at least one of the dimensions.*

Without loss of generality, we will assume that "better" means "smaller". Therefore, we say that a point p dominates q (and we write $p \prec q$) when $\forall i \in [1, d]$, $p[i] \leq q[i]$ and $\exists j : p[j] < q[j]$, where d is the total number of dimensions and $p[i]$ is the value of p in the i-th dimension. The concept of domination leads naturally to the concept of *skyline* [5].

DEFINITION 2 (SKYLINE QUERY). *The result of a skyline query, $SKY(\mathcal{S})$ over a dataset $\mathcal{S}$ is composed of all points that are not dominated by any other point. Formally:*

$$SKY(\mathcal{S}) = \{p \in \mathcal{S} : \nexists q \neq p \ s.t. \ q \prec p\}$$

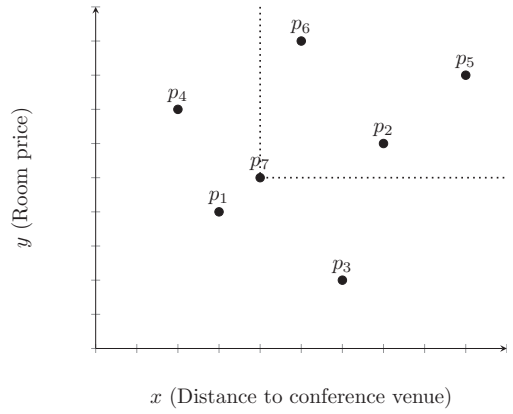

Figure 1: The hotel database.

Essentially, the skyline set contains the best possible objects with respect to the attribute values. This means that there is no other point which is strictly better that the points in $SKY(\mathcal{S})$ in any of the dimensions. Evidently, the skyline query may be combined with additional constraints on the attribute values. For example, we may ask for the skyline set of a small area of the data space and not of the entire dataset. Moreover, the skyline set is invariant in dimension scaling and it does not require any user-defined function, since it is based solely on the concept of domination. However, the cardinality of $SKY(\mathcal{S})$ depends heavily on the data distribution and dimensionality, resulting sometimes in cumbersome results (too many skyline points) and sometimes in very few.

On the contrary, a top-k query [9] returns exactly k points which are the best in terms of a user-defined scoring function. Most of the algorithms assume that the scoring function is monotone with respect to the values of the dimensions.

Definition 3 (TopK Query). *The result of a top-k query over a dataset $\mathcal{S}$, with respect to a specific scoring function, is composed of the k points with the maximum scores.*

An example is given in Figure 1. The result of the skyline query is composed of the points p_1, p_3 and p_4, since these points are not dominated by any other point of the dataset. In contrast, the rest of the points are dominated by at least one skyline point. For instance, p_7 is dominated by p_1 whereas p_2 is dominated by both p_1 and p_3. The result of a top-2 query based on the scoring function $f(p) = p.x + p.y$ (sum of coordinates) consists of p_1 and p_3. We observe that p_1 is contained in both the skyline query result and the top-2 query result. In fact, it can be proved that for any monotone scoring function, the top-1 query result is always a member of the skyline.

As an attempt to eliminate the limitations of skyline and top-k queries another preference-based query type has been proposed initially in [17] and later studied in detail in [20, 21]. As in a typical top-k query, the k objects with the highest scores are selected and returned to the user. However, a fundamental difference is that no user-define scoring function is required since by default the dominance relation is being used. This is done by defining the score of an object p to be the number of objects that p dominates.

Definition 4 (Dominance Score). *The dominance score $s(p)$ of a point p is the number of points dominated by p. Formally:*

$$s(p) = |\{q \in \mathcal{S} | p \prec q\}| \quad (1)$$

Definition 5 (Top-k Dominating Query). *The result of a top-k dominating query consists of the k points with the highest scores with respect to domination.*

Top-k dominating queries have the following desirable properties:

- the result does not depend on the scaling of the dimensions,
- no additional scoring functions are required and
- the cardinality of the result is controlled by the parameter k.

A top-2 dominating query on the data set of Figure 1 will return the points p_1 and p_7. Note that, p_1 dominates four points (p_2, p_5, p_6 and p_7) and p_7 dominates three points (p_2, p_5 and p_6).

Related Work. A basic method for retrieving the top-k dominating points of a dataset would consist of, firstly, computing the dominance score of each point and then using a linear time selection algorithm [3] to find the point v with the k-th largest score. To find all the top-k dominating points we perform a final scan on the dataset and report all points with a greater score than the score of v.

The simplest approach for computing the domination score for all points would be to compare each point p with every other point q in the dataset and increment p's score if it dominates q. This results in $O(n^2)$ time cost and $O(n)$ space cost. An approach with lower time complexity would be to use a 2-dimensional range counting data structure (e.g., [8, 12]). For each point $p = (x_p, y_p)$ in a dataset $\mathcal{S}$, the points lying in the query rectangle $Q = [x_p, \infty) \times [y_p, \infty)$ can be counted in $O(\log n)$ time and $O(n)$ space using the 2-dimensional range counting data structure by Chazelle [8]. The number of points found in Q is equal to p's domination score. In order to compute the dominance score

of each point, we repeat the process for all the points in $\mathcal{S}$ in $O(n \log n)$ total time. Lastly, an algorithm by Chan and Pătraşcu [7] is able to compute the dominance score for all points in $O(n\sqrt{\log n})$ time in the word-RAM model [11] of computation. Insertions and deletions can be trivially supported in the above methods in $O(n)$ time since one has to update the dominance scores of all points in the worst-case. In the following, we describe more elaborate methods to answer a top-k dominating query.

Papadias et al. [17], first proposed the d-dimensional top-k dominating query along with a solution based on the iterative computation of a dataset's skyline points. More specifically, they observed that the top-1 dominating point of a dataset is contained in the dataset's skyline points. This stems from the observation that for every point p not in the skyline, there exists a point p' in the skyline that dominates it and, as a result, p' has a larger score than p. Thus, in their approach, they compute the set of skyline M (using the BBS algorithm [17]) and compute the dominance score of all the points in M. The point q with the highest score is the top-1 dominating point and is thereby reported. Finally, q is removed from the dataset and the procedure is repeated until k points have been reported. However, this approach does not avoid the quadratic trap, since the score computation of skyline points as well as the update of dominance scores after the removal of the point with the highest score, may lead to $O(n^2)$ dominance checks, whereas the space remains linear [1].

Yiu and Mamoulis [20, 21] recommended using aggregate R-trees (aR-trees) to efficiently compute d-dimensional top-k dominating queries. They provided various algorithms based on aR-trees that proved experimentally to be quite fast. They also make an analytic study making the assumption that the data points are uniformly and independently distributed in a domain space. The authors do not make any statement for the worst-case time complexity of the query but it is certainly $\Omega(n)$.

Both methods [17, 21] focus on the top-k dominating query, where k is arbitrary. Update operations can be applied in both cases with a linear time cost. However, the top-k dominating query has to be re-evaluated in both cases. Finally, both prove the efficiency of their approach experimentally (extensive experiments can be found in [21]).

As a closing remark, we should further note that top-k dominating queries have also been studied in the context of uncertain databases [14, 22], data streams [13], spatial objects [19] and vertically decomposed data [18].

Motivation, Contribution and Assumptions. This work is the first attempt to provide efficient algorithms for top-k dominating query processing in the semi-dynamic and the fully-dynamic cases, which are the most interesting and challenging. In contrast to previously proposed techniques, we are interested in algorithms with non-trivial performance guarantees.

One may think that perhaps a direct application of the divide-and-conquer algorithmic technique could provide an efficient solution at least for the static top-k dominating query, where given the dataset $\mathcal{S}$ we are asking for the k points with the highest domination scores. The problem with this approach is that the top-k dominating query is a *non-decomposable query*, because the score of each point depends on the coordinates of all the other points in $\mathcal{S}$. A query q in $\mathcal{S}$ is decomposable [2] if its output can be computed accurately by executing q in a partition of $\mathcal{S}$. The non-decomposability of top-k dominating queries prohibits us from using standard divide-and-conquer techniques and thus increases the problem difficulty significantly.

This paper concentrates on 2-dimensional data for two reasons. First, there is no previous work with asymptotic guarantees and as a result, this paper provides a deeper understanding of the complexity of the problem. The second, more practical, reason is that many applications are inherently 2-dimensional. This is because, one often faces the situation of having to strike a balance between a pair of naturally contradicting factors (e.g., price vs quality, space vs query time). Finally, our algorithms are based on a novel restricted dynamization of layers of minima [4]. This is of independent interest in case we only need to access the first k layers of minima.

Since static datasets are being handled rarely by modern applications, we consider the problem in the semi-dynamic case (insertions only), where logarithmic complexities are attained. In the fully-dynamic case, we attain polynomial complexities for update operations (insertions and deletions). In many applications, insertions occur much more frequently than deletions. As a practical example, consider an application that retrieves the top-k dominating tweets (i.e., Twitter messages) according to some user-selected attributes. In this application, the semi-dynamic algorithms would suffice since tweets very rarely are deleted [1]. Other possible examples of datasets where insertions take place significantly more frequently than deletions include the measurements collected by a scientific instrument or the full-year sales log of a retail company. In conclusion, applications where deletions occur orders of magnitude less frequently than insertions can benefit from the use of the semi-dynamic algorithms and the associated data structures.

For each of the semi-dynamic and fully dynamic settings we provide two solutions (k-list and 1-list) that provide a trade-off between update and query time. All

[1]The BBS algorithm is based on the use of R-trees which require linear space.

Table 1: SD stands for Semi-Dynamic, where only insertions are allowed, whereas FD stands for Fully-Dynamic where both insertions and deletions are supported.

Algorithm	Space	Preprocessing Cost (worst-case)	Query Cost (worst-case)	Update Cost (amortized)
SD/k-list	$O(n)$	$O(n \log n)$	$O(k)$	$O(\log^2 n + k^2 \log n)$
SD/1-list	$O(n)$	$O(n \log n)$	$O(k \log n)$	$O(\log^2 n + k \log n)$
FD/k-list	$O(n)$	$O(n \log n)$	$O(k)$	$O((k + \sqrt{n})k \log n)$
FD/1-list	$O(n)$	$O(n \log n)$	$O(k \log n)$	$O((k + \sqrt{n}) \log n)$

our algorithms use linear space and work well under the realistic assumption that k is a fixed user-defined parameter which is small compared to the size n of the dataset (i.e., $k << n$). Table 1 provides a detailed overview of our results.

Roadmap. The rest of the paper is organized as follows. Section 2 presents some necessary concepts related to the discussion that follows. Our contribution for the semi-dynamic case is detailed in Section 3, whereas the study of the fully-dynamic case is offered in Section 4. In addition to the results for the RAM computation model, in Section 5 we provide an adaptation to the word-RAM model, obtaining better asymptotic bounds. Concluding remarks and directions for further research are offered in Section 6.

2. PRELIMINARIES

In this section, we discuss the basic concepts that are used throughout the rest of this work. First of all, we note that we augment the definition of each point p_i to also include its score $s_i = s(p_i)$, so p_i becomes a triple of the form $p_i = (x_i, y_i, s_i)$.

In the following two sections, we describe the concept of *layers of minima* and we cite a previous result in the form of a lemma, that will be used in the query phase of some of our proposed solutions.

2.1 Two-dimensional Layers of Minima

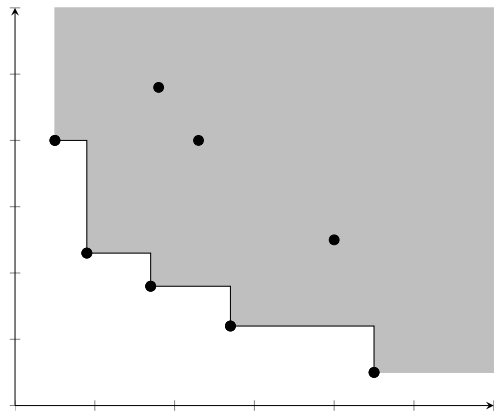

Figure 2: The first layer of minima (skyline).

The algorithms presented in the remaining sections are based on the concept of *layers of minima*. In order to compute the layers of minima of a dataset $\mathcal{S}$ we perform a skyline query on $\mathcal{S}$, remove the answer set of points from $\mathcal{S}$ and repeat the process until no points remain in $\mathcal{S}$. The set that results from the i-th skyline query forms the i-th layer of minima. By collecting all the layers, we form the layers of minima of $\mathcal{S}$. A concise definition of the layers of minima follows:

DEFINITION 6. *Let M_1 be the set of all minima points in $\mathcal{S}$. The first layer of minima of $\mathcal{S}$ is equal to the set M_1 and the second layer of minima M_2 of $\mathcal{S}$ is equal to the set of all minima points in $\mathcal{S} - M_1$. The j-th layer of minima of $\mathcal{S}$ is accordingly defined to be equal to the set of all minima points in $\mathcal{S} - (\bigcup_{i=1}^{j-1} M_i)$. The set $\{M_1, M_2, \ldots, M_\lambda\}$ where $\mathcal{S} = \bigcup_{i=1}^{\lambda} M_i$ is the layers of minima of $\mathcal{S}$.*

Figure 2 depicts a layer of minima on the plane. Any point located in the shadowed region is dominated by at least one point in the layer of minima.

Blunck and Vahrenhold [4] proposed in-place algorithms that use $O(1)$ extra space and compute the layers of minima of a dataset of 2-dimensional points in $O(n \log n)$ time.

2.2 Reporting Lemma

Finally, we use the following lemma from [10]:

LEMMA 1. *Let $A_1, \ldots, A_m$ be arrays of values from a totally ordered set such that each array is sorted. Given an integer $L \leq \sum_{i=1}^{m} |A_i|$, there is a comparison-based algorithm that finds in $O(m)$ time a value τ that is greater than at least L but at most $O(L)$ values in $A_1 \cup \ldots \cup A_m$.*

Lemma 1 can be adjusted to report a value τ that is smaller than at least L but at most $O(L)$ values in $A_1 \cup \ldots \cup A_m$. This lemma forms the basis in allowing us to efficiently find the k-th point with the highest score out of a collection of ordered lists and is used in Sections 3 and 4.

3. THE SEMI-DYNAMIC CASE

In this section, we propose a solution to the semi-dynamic top-k dominating query problem and describe

in detail the data structures and algorithms we use to achieve it. Let $\mathcal{S}$ be a set of n 2-dimensional points. Recall that the semi-dynamic top-k dominating query aims at reporting the k points in $\mathcal{S}$ with the highest dominance score where k is a fixed user-defined parameter. Furthermore, $\mathcal{S}$ is subject to insertions of new points. This poses an additional challenge since after inserting a new point, it is possible that the dominance score of many (or even all) the points in $\mathcal{S}$ must be updated. Individually updating the score of each such point would be computationally prohibitive so we follow a different approach and only update lazily the score of groups of points that are candidates for being in the final answer.

We first note that when a point p dominates another point q, p's score is strictly greater than the score of q:

$$\forall p, q \in \mathcal{S}, p \prec q \Rightarrow score_p > score_q \tag{2}$$

Organizing $\mathcal{S}$ into layers of minima offers an intuitive way of using the above property to eliminate points that are not possible to belong in the final answer. As an example, consider a top-1 dominating query in $\mathcal{S}$. The point with the highest dominance score is found in the first layer of minima of $\mathcal{S}$ since all the points in the second and subsequent layers are dominated by at least one other point. Similarly, in a top-2 dominating query, the first point is found in the first layer and the second point is found in either the first or the second layer. In general, the following lemma holds for the top-k dominating points:

LEMMA 2. *The top-k dominating points of $\mathcal{S}$ are located in the first k layers of minima of $\mathcal{S}$.*

PROOF. If $\mathcal{S}$ has only k or less layers of minima, the lemma obviously holds. Otherwise, assume that a point p belongs to the i-th layer of minima, where $i \geq k+1$. There are at least $i-1$ points dominating p and due to Equation 2 all of them have a larger score than p. As a result, p is not included in the top-k dominating points of $\mathcal{S}$. □

A direct consequence of Lemma 2 is that, when inserting a new point p, we only need to update the scores of some points in the first k layers of minima. However, some of the layers may have many points and thus individually updating the score of these points would result in a high update cost. To avoid this, after inserting a new point p in $\mathcal{S}$, we find only the first and last point that dominate p in each layer. This pair of points denotes an interval that marks all the points in each layer whose score must be updated. Consequently, by examining only $O(1)$ points in each layer the total update cost is reduced.

Lastly, an issue brought up by the use of layers of minima is that the insertion of a new point p may create cascading changes to the structure of the layers. In particular, by inserting p into $\mathcal{S}$, p must also be inserted in one of the layers of $\mathcal{S}$. Let L_i be that layer. The insertion of p in L_i may cause some of its points to be discarded as a result of them being dominated by p. This group of points must be inserted into the next layer L_{i+1} possibly discarding some of the points in L_{i+1} in the process. Due to Lemma 2 and the fact that only insertions are allowed, this chain of operations only has to be performed up until the k-th layer.

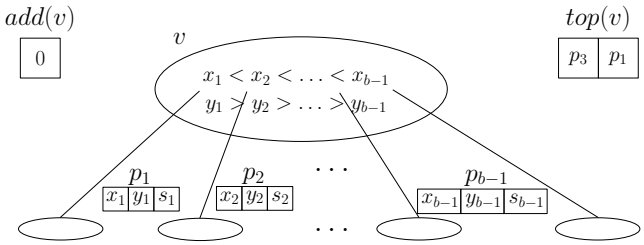

Figure 3: An (a,b)-tree node ($k=2$). The coordinates in node v are designated by the respective representative points p_i, $1 \leq i \leq b-1$, where p_i is the leftmost point of the $i+1$-th child of v.

To achieve efficient insertion, we model each layer as an (a,b)-tree. In the following, we provide a detailed overview of the data structure and the operations it supports and then we describe the update and query algorithms for the semi-dynamic top-k dominating query.

3.1 The Augmented (a,b)-Tree

We model each layer of minima using an augmented leaf-oriented (a,b)-tree. Assume that L is a layer of minima containing m points, i.e., $p_1, p_2, \ldots p_m$ where $p_i = (x_i, y_i, s_i), 1 \leq i \leq m$. Since the points in L are totally ordered on each dimension[2], we can use a single (a,b)-tree to search among the points in both dimensions. To achieve that, each inner node stores representative keys for both dimensions, instead of storing keys for only one of them.

For each node v of the tree with $height \geq \log_b k$ we maintain a field $add(v)$. The field's contents denote a score that has to be added to the score of all the points in v's subtree. Finally, each node v with $height \geq \log_b k$ is augmented with an k-sized list $top(v)$ which stores the k points with the highest score in v's subtree. An example of an augmented (a,b)-tree node with $height > \log_b k$ for $k=2$ can be seen in Figure 3.

For the remainder of this work we assume that $b = O(1)$ since we present main memory algorithms. The following lemma provides the tree's total space cost.

LEMMA 3. *The total space required by an augmented (a,b)-tree storing m points is $O(m)$.*

[2]For two points $p_a = (x_a, y_a, s_a)$ and $p_b = (x_b, y_b, s_b)$ in L if $x_a > x_b$ then $y_b > y_a$

PROOF. All the nodes with height lower than $\log_b k$ only store $O(1)$ additional information so their total space cost is $O(m)$. There are $O(m/k)$ nodes with height higher than or equal to $\log_b k$ each augmented with a k-sized list. The total space cost of this part of the data structure is $O(m/k) \times O(k) = O(m)$. As a result, the total space cost of the entire data structure is $O(m)$. □

The following lemma provides the time complexity for the construction of an augmented (a, b)-tree.

LEMMA 4. *The construction of an augmented (a, b)-tree over m points that are sorted according to their dimensions can be carried out in $O(m \log k)$ time, where k is a user-defined parameter.*

PROOF. In order to construct the leaf-oriented augmented (a, b)-tree we follow a bottom-up approach and assume that the input points are sorted according to their dimensions. The augmented (a, b)-tree is constructed in a similar way to a typical (a, b)-tree with an additional issue. At first, the nodes of the augmented (a, b)-tree are constructed by scanning the input points, creating the leaves and then recursively creating the inner nodes from bottom to top. Each node is only visited once so the procedure up to this point requires $O(m)$ time.

The last step is to compute the *top* lists for all nodes with $height_v \geq \log_b k$. For each node v with $height_v > \log_b k$, the $top(v)$ list must be computed from the *top* lists of v's children. By simultaneously traversing the $O(b) = O(1)$ *top* lists of v's children we can compute $top(v)$ in $O(k)$ time. There are $O(m/k)$ nodes with $height_v > \log_b k$ and since this process is repeated for every node, the time required is $O(m/k) \times O(k) = O(m)$.

Finally, we compute the *top* lists for each node v with $height_v = \log_b k$. Since v's children are not augmented with *top* lists we follow a different approach. We sort all the points found in v's subtree[3] in $O(k \log k)$ time and store them in $top(v)$. There are $O(m/k)$ nodes with $height_v = \log_b k$ and thus this step requires $O(m \log k)$ total time.

Thus, the total time required for the construction of the (a, b)-tree is $O(m \log k)$. □

3.1.1 Operations

In this section, we formally describe all the operations supported by the augmented (a, b)-tree. More specifically, the augmented (a, b)-tree supports searching for a point, inserting a new point, or deleting an existing one. Furthermore, splits and concatenations between two different (a, b)-trees are also supported.

[3]There are up to k points in v's subtree since $height_v = \log_b k$

The search operation `search`(T, p_z) locates in the augmented (a, b)-tree T a specific point p_z and can be performed with respect to either dimension of p_z by using the appropriate set of keys. Let v be a node of T, $x_1, x_2, \ldots, x_{b-1}$ be the x-representative keys of v's children and $y_1, y_2, \ldots, y_{b-1}$ be the y-representative keys of v's children. In order to search for a point $p_z = (x_z, y_z, s_z)$ in T, we begin at the root and search down until we reach a leaf. If the search is performed on the x dimension, we select the i-th child of v such that $x_{i-1} < x_z \leq x_i$. Otherwise, if the search is performed on the y dimension, we select the i-th child of v such that $y_{i-1} > y_z \geq y_i$. Since T is height-balanced, a search operation requires $O(b \log m) = O(\log m)$ time.

The rest of the operations are based on node splits and merges. For reasons of clarity, we first describe how node splits and node merges are handled on the augmented (a, b)-tree in relation to typical (a, b)-trees.

The node split operation `node_split`(v, v_1, v_2) is performed similarly to the split operation of typical (a, b)-trees with a few modifications. More specifically, before dividing a node v into two nodes v_1 and v_2 we check the contents of $add(v)$. If $add(v)$ stores a value different than 0, we add the contents of $add(v)$ to the *add* variable of v's children and set $add(v)$ to 0. Afterwards, v is divided into v_1 and v_2 and the keys for the x and y dimensions of v are "shared" between v_1 and v_2 in $O(b) = O(1)$ time. After sharing the keys, $top(v_1)$ for v_1 and $top(v_2)$ for v_2 must be recomputed. If $height_v > \log_b k$ we can compute $top(v_1)$ and $top(v_2)$ in $O(k)$ time by simultaneously traversing the $O(b) = O(1)$ *top* lists of v_1's and v_2's children respectively. As a result, the split operation requires $O(k)$ time in this case.

If $height_v = \log_b k$ then the children of v_1 and v_2 are not augmented with *top* lists and thus the computation of $top(v_1)$ and $top(v_2)$ cannot be performed using the above procedure. Each of v_1 and v_2 have up to k points in their subtree since their height is equal to $\log_b k$. To compute the *top* lists for v_1 and v_2 we sort all the points in v_1's and v_2's subtree in $O(k \log k)$ time and store the result in $top(v_1)$ and $top(v_2)$ respectively. This results in the split operation requiring $O(k \log k)$ time in this case. Finally, if $height_v < \log_b k$ then the split operation requires $O(b) = O(1)$ time since v is not augmented with $top(v)$.

For the merge operation `node_merge`(v_1, v_2, v) we follow a similar procedure to the merge operation of standard (a, b)-trees. More specifically, before merging two nodes v_1 and v_2 into v we check the contents of $add(v_1)$ and $add(v_2)$. If $add(v_1)$ stores a value different than 0 we add the contents of $add(v_1)$ to v_1's children and set $add(v_1)$ to 0. We follow the same procedure for $add(v_2)$. Then, v_1 and v_2 are merged into v and the keys for the x and y dimensions of v are derived from the keys of v_1 and v_2 in $O(b) = O(1)$ time. After merging the

keys, $top(v)$ must be recomputed. To achieve this, if $height_v > \log_b k$ we simultaneously traverse $top(v_1)$ and $top(v_2)$ and store the k points with the highest score in $top(v)$ in $O(k)$ time. If $height_v = \log_b k$ we follow a similar approach to that of a node split and first compute $top(v_1)$ and $top(v_2)$ by sorting all the points in v_1's and v_2's subtree respectively. Then by simultaneously traversing $top(v_1)$ and $top(v_2)$ we store the k points with the highest score in $top(v)$. As in the split operation, if $height_v < \log_b k$ the merge operation requires $O(b) = O(1)$ time due to the fact that v is not augmented with $top(v)$.

Operation `insert`(T, p), inserts a point p in T. The point is inserted as a leaf in T and the tree is rebalanced using node splits. Since there are $O(\log m)$ node splits that cost $O(k)$ time and $O(1)$ splits that cost $O(k \log k)$ time, the time cost to insert a point is $O(k \log m)$.

Operation `delete`(T, p), removes a point p from T. The leaf corresponding to the point is removed and the resulting tree is rebalanced. There are $O(\log m)$ merges that cost $O(k)$ time, $O(1)$ merges that cost $O(k \log k)$ time and a possible terminating split and as a result the time cost to delete a point is $O(k \log m)$.

Using the node split and node merge operations as building blocks, we can define two additional operations on the augmented (a, b)-trees: Tree Concatenation and Tree Split. For both the operations, we use the definition and algorithms provided in [15].

Operation `concat`(T_1, T_2, T_3), concatenates two augmented (a, b)-trees T_1 and T_2 into a third augmented (a, b)-tree T_3. This operation assumes that $\max\{T_1\} \leq \min\{T_2\}$ where $\max\{T_i\}$ is the largest x coordinate of all the points in T_i and $\min\{T_i\}$ is smallest x coordinate of all the points in T_i (a similarly defined order is implied for the y dimension as well). In a tree concatenation one merge operation and up to $O(\log \max(|T_1|, |T_2|))$ split operations are performed. Since there are $O(1)$ merge and split operations that cost $O(k \log k)$ time and the rest of the merge and split operations cost $O(k)$ time, a tree concatenation operation requires $O(k \log \max(|T_1|, |T_2|))$ time. Before initiating this operation, all *add* variables in the affected path are flashed to their children (get a zero value).

Operation `split`(T_1, val, T_2, T_3), splits an augmented (a, b)-tree T_1 into two augmented (a, b)-trees T_2 and T_3 at element val with respect to the one of the two dimensions, so that $T_2 \leftarrow \{z \in T_1; z \leq val\}$ and $T_3 \leftarrow \{z \in T_1; z > val\}$. In a tree split operation the starting (a, b)-tree is first split into two forests of trees. Then, the roots of the trees in each forest are merged with each other recursively. Splitting the tree into two forests requires $O(\log |T_1|)$ time and since there are $O(1)$ merge operations that cost $O(k \log k)$ time and $O(\log |T_1|)$ merges for both forests, each requiring $O(k)$ time, a tree split operation requires $O(k \log |T_1|)$ time. Similarly to `concat`, before initiating this operation, all *add* variables in the affected path are flashed to their children. The following theorem summarizes the discussion on the (a, b)-tree.

THEOREM 1. *Given a parameter k and m 2-dimensional points $p_i = (x_i, y_i, s_i)$ where $1 \leq i \leq m$, we can construct in $O(m \log k)$ time an augmented (a, b)-tree T_1 that uses $O(m)$ space. The construction time assumes that the points are sorted according to their dimensions. The tree T_1 supports the following operations:*

- *`search`(T_1, p) in $O(\log m)$ time,*
- *`insert`(T, p) and `delete`(T, p) in $O(k \log m)$ time,*
- *`split`(T_1, val, T_2, T_3) in $O(k \log m)$ time and*
- *`concat`(T_1, T_4, T_5) in $O(k \log \max(|T_1|, |T_4|))$ time where T_4 is an augmented (a, b)-tree such that the condition $\max\{T_1\} \leq \min\{T_4\}$ holds.*

3.2 Insertion

Let $p = (x_p, y_p, s_p) \in \mathrm{R}$ be a point to be inserted into $\mathcal{S}$. Furthermore, let $L_1, \ldots, L_k$ be the first k layers of minima of $\mathcal{S}$. Before inserting p we compute its dominance score using the dynamic range counting data structure proposed in [8][4]. We also insert p in the dynamic range counting data structure in order to support score computation for future insertions. The data structure supports queries and updates in $O(\log^2 n)$ time and $O(n)$ space.

Afterwards, we find if p must be inserted in one of $L_1, \ldots, L_k$ by searching each of the k respective (a, b)-trees for p. Starting from L_1 and iterating towards L_k, we search each tree for p both in the x and in the y dimension and retrieve the predecessor of p in the x dimension and the predecessor of p in the y dimension. If neither of those two points dominate p, we insert p in the tree's respective layer and stop the iteration. Otherwise, the iteration may end without any layer satisfying the above condition. In that case, p does not become a member of any of the k first layers.

If we do not insert p in any of the k first layers then we only have to update the scores of some points in each of $L_1, \ldots, L_k$. Otherwise, assume that p is inserted into L_i where $1 \leq i \leq k$. Then we have to update scores of points in $L_1, \ldots, L_{i-1}$ and alter the structure of $L_i, \ldots, L_k$ (Figure 4). We first describe how to handle *score updating* on a layer and afterwards how to *alter a layer's structure* using tree splits and tree concatenations.

To update the score of the points in layer L, we perform this procedure. We search the augmented (a, b)-tree of L for x_p and y_p. All points whose score must be updated lie to the left of x_p and to the right of y_p. Since

[4]The data structure is built only once as a preprocessing step before the first insertion

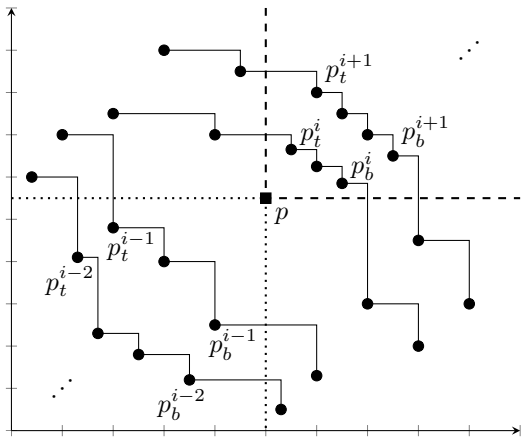

Figure 4: Insertion of a new point p.

updating the score of each point would be time consuming, we only find the two boundary points that define the above interval and mark the subtrees between them. For example, in Figure 4 the insertion of p causes a score increment for the points p_t^{i-2} to p_b^{i-2} in layer L_{i-2} and the points p_t^{i-1} to p_b^{i-1} in layer L_{i-1}.

We start from $height = \log_b k + 1$ of the two search paths and move up towards the root, adding $+1$ to $add(v)$ if v is a node hanging to the left of the search path for x_p or to the right of the search path for y_p. Using this method we denote that the score of all the points in v's subtree must be incremented by one, without actually visiting the points themselves. Adding $+1$ to $add(v)$ does not change $top(v)$ since we increment the score of all the points in v's subtree and thus their relative order according to score remains unchanged. For each node v' on the search paths or hanging on the search paths with $height_{v'} = \log_b k$, instead of incrementing $add(v')$, we exhaustively check the points in the subtree of v' and individually update their score based on if they are dominating p. Finally, we sort the points in the subtree of v' based on the updated scores and store the result in $top(v')$. For each node with $height < \log_b k$ no action is necessary since all the points in its subtree can be found in the *top* list of its ancestor with $height = \log_b k$. Thus, at the end, we have indirectly marked all the points between y_p and x_p for score increment.

Finally, we update the *top* lists of the nodes in the search path as a result of modifying the *add* fields of their children. Starting from $height = \log_b k + 1$ and moving towards the root, we recursively compute the *top* list of each node v by simultaneously merging the *top* lists of its children. While merging the lists, we also add the contents of each node's *add* field to the score of the node's *top* list points so as to take into account the score changes caused by the insertion of p. At the end,

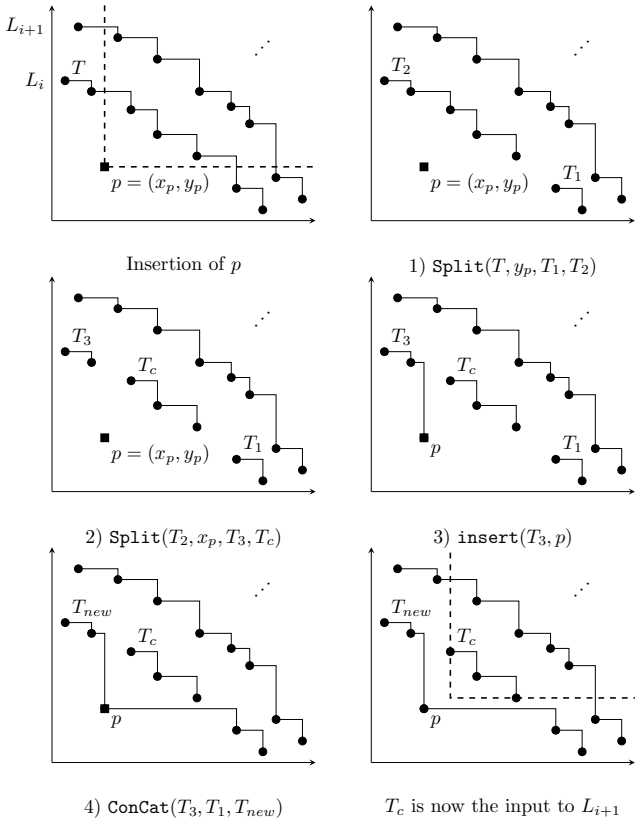

Figure 5: Example of layer restructuring.

the *top* list found in the root of the (a, b)-tree will have the correct top-k points for that layer of minima.

Indirectly marking all the points between y_p and x_p for score increment requires $O(\log n + k \log k)$ total time while merging the top-k lists of a node's children requires $O(bk) = O(k)$ time and as a result the total cost for all the nodes in the search paths of the tree is $O(bk \log n) = O(k \log n)$ time. Thus, the total time required to update scores in a layer is $O(k \log n)$.

In the second case, the point p may have to be inserted in layer of minima L_i. Since inserting or deleting points from the layer of minima one-by-one would be time consuming, we insert the point and remove the now-dominated points by executing a series of tree splits and tree concatenations. First, we find the interval of points dominated by p as previously by querying the layer of minima tree T for x_p and y_p. Then we perform the following sequence of operations in order:

1. `split`(T, y_p, T_1, T_2),
2. `split`(T_2, x_p, T_3, T_c),
3. `insert`(T_3, p) and
4. `concat`(T_3, T_1, T_{new}).

Recall that all *add* variables of nodes on the affected paths for operations `split` and `concat` are flashed to

their children, that is in all such paths the *add* variables have zero value. The effects of this sequence of operations on the layers of minima can be seen in Figure 5. The layer of minima tree T_{new} for L_i now correctly has p inserted and every point previously in L_i that is now dominated by p (i.e., T_c) has been discarded.

If T_c is empty, the iteration stops. Otherwise, T_c is propagated to the next layer of minima L_{i+1} where we repeat the above procedure with T_c as the input. Since T_c may have more than one points, instead of inserting them one-by-one we perform a tree concatenation at step (3) instead of an insertion. Finally, the insertion spot of T_c in L_{i+1} can be found by querying the tree of L_{i+1} for $p' = (x'_p, y'_p)$ where x'_p is the x coordinate of the leftmost point in T_c and y'_p is the y coordinate of the rightmost point in T_c.

In Figure 4, the insertion of p causes the points p_t^i to p_b^i to be discarded from layer L_i. These points are propagated to layer L_{i+1} and will, in turn, discard the points p_t^{i+1} to p_b^{i+1} from L_{i+1}. This procedure is repeated until the k-th layer of minima.

In each layer of minima we perform a series of $O(1)$ splits and concatenations. Since each tree split or concatenation requires $O(\log n)$ time, the total time required to alter a layer's structure given a point or an (a, b)-tree as an input is $O(k \log n)$.

As described in the beginning of the section, after an insertion a layer must either update the score of some of its points or alter its structure. Since either case requires $O(k \log n)$ time, the time cost of manipulating the k first layers after an insertion is $O(k^2 \log n)$. Adding the cost of computing the score of the inserted point and inserting the point in the dynamic range counting data structure, the total insertion cost is $O(\log^2 n + k^2 \log n)$ amortized time.

3.3 Query

To find the top-k dominating points of $\mathcal{S}$, we apply Lemma 1 for $L = k$ on all the top-k lists found in the root of each (a, b)-tree of each of the k first layers of minima. Let I be the list returned by Lemma 1. By selecting the $(|I| - k + 1)$-th order statistic of I we obtain the dominance score τ of the k-th top dominating point. Finally, we traverse all the top-k lists we previously collected and report all points with score larger than τ. Since the lists are sorted according to their score, we can stop traversing a list when a point with score lower than τ has been found. Applying Lemma 1 requires $O(k)$ time while finding the $(|I| - k + 1)$-th order statistic of I requires $O(I) = O(k)$ time. Finally, traversing all lists requires $O(k)$ time in total. By combining all of the above, we achieve $O(k)$ query time.

3.4 Reducing the Update Cost

We can reduce the algorithm's update cost by shrinking the size of the *top* list in each node of each (a, b)-tree. In particular, we store a *top* list in each node of the (a, b)-tree but instead of storing k points in each list we only store 1. This removes the cost of computing *top* lists during each node split or merge since each *top* list can be computed using $O(b) = O(1)$ comparisons. As a result, node splits and merges cost $O(1)$ time and updating the score of points in a layer or altering its structure costs $O(\log n)$ time. This brings the total insertion cost down to $O(\log^2 n + k \log n)$ amortized time.

This change also implies that at the time of a query, each (a, b)-tree's root only stores 1 element with the highest score in the layer and as a result we can no longer directly apply Lemma 1. To overcome this we build a Strict Fibonacci Heap [6] by inserting each point with the highest score from each layer. Strict Fibonacci Heaps support insertions in $O(1)$ worst-case time and deletions of the maximum key in $O(\log n)$ worst-case time. By querying the heap we are able to find (and delete) the top-1 dominating point. After deleting a point p (belonging in a layer L) from the heap, we have to replace it by the point of L with the next highest score. This point can be found by querying L's tree for p. Due to the definition of *top* lists, the point with the next highest score in L is guaranteed to be amongst the $O(b)$ *top* lists of each node in the search path of p. We insert all $O(b \log n) = O(\log n)$ such points in the heap and repeat the process until k points have been deleted from the heap. In order to not have any duplicate points in the heap, we also employ a marking process.

Deleting a point from the heap requires $O(\log n)$ time, while adding $O(\log n)$ points also requires $O(\log n)$ time. Since there aren't any duplicate points in the heap and the process is repeated k times, the query phase of the algorithm requires $O(k \log n)$ time.

Lastly, we review the preprocessing cost in both semi-dynamic algorithms. In the case of the k-list augmented (a, b)-tree the construction time is equal to $O(n \log n)$. To achieve this, we build Chazelle's static range counting data structure [8] in $O(n \log n)$ time and count the score of each point using the method we described in Section 1. We also build Chazelle's dynamic range counting data structure [8] which is required by our insertion algorithm in $O(n \log n)$ time. The layers of minima can be computed in $O(n \log n)$ time [4]. A subsequent scan of the output provides us with the points of each layer of minima in sorted order and as a result we can build the (a, b)-trees in $O(n \log k)$ time (Lemma 4). Therefore, the construction time is equal to $O(n \log n)$. In the case of the 1-list augmented (a, b)-tree the only difference is the construction cost of the (a, b)-trees which is reduced to $O(n)$ since each node in each (a, b)-tree is augmented with a *top* list of size 1. The discussion of this section can be summarized in the following theorem:

THEOREM 2. *Given a set of n 2-dimensional points, we can build a data structure that supports insertions of new points in $O(\log^2 n + k^2 \log n)$ amortized time and top-k dominating queries in $O(k)$ worst-case time. Alternatively, we can build a data structure that supports insertions of new points in $O(\log^2 n + k \log n)$ amortized time and top-k dominating queries in $O(k \log n)$ worst-case time. Both data structures are built in $O(n \log n)$ time and use $O(n)$ space.*

4. THE FULLY-DYNAMIC CASE

The algorithms presented so far only support insertions due to the fact that all operations could be restricted in the first k layers of minima of a dataset $\mathcal{S}$. However, assume the deletion of a point p in layer L_k. Then we would have to store and manipulate more than k layers since it is possible that some points from L_{k+1} might have to be inserted in L_k as a result of them not being dominated by any other point in L_k apart from p. This brings a cascading of restructuring operations since some of the points in L_{k+2} might have to be inserted in L_{k+1}. Thus, a deletion operation may reach the last layer of $\mathcal{S}$ in the worst-case. It should be noted that a deletion of a point may not always result in layer restructuring. Consider the example in Figure 6. Deleting p_d will cause the layers to be restructured since p_f is not dominated by any other point. However, deleting p_e will not cause any changes to the layers' structure since both p_g and p_h are dominated by at least one other point in p_e's layer.

A deletion of an existing point can be defined in a similar way to the insertion of a point with each layer requiring either score updating or restructuring. To perform score updating we follow the same steps as those discussed in Section 3.2 but instead of adding $+1$ to the *add* field of a node, we add -1. After deleting a point $p = (x_p, y_p)$ from a layer L_i, we query L_{i+1} to find all the points (if any) that must be inserted in L_i due to the deletion of p. The query point in L_{i+1} is $p' = (x'_p, y'_p)$ where x'_p is the successor of x_p in L_i and y'_p is the successor of y_p in L_i.

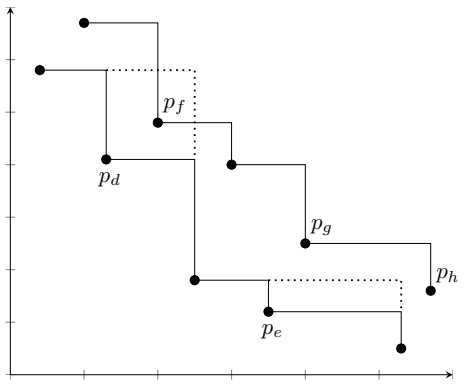

Figure 6: Deletion of existing points.

Our algorithms for the semi-dynamic setting can be extended to the fully dynamic setting through the use of the global rebuilding technique [16]. More specifically in an update operation, instead of manipulating only the first k layers we perform score updates and layer restructuring operations in the first $k+\sqrt{n}$ layers. Since we stop restructuring operations on a predefined point, after the i-th deletion the $(k+\sqrt{n}-i+1)$-th layer will have become invalid. As a result, after $\sqrt{n}$ deletions, only the first k layers remain valid and at that point we rebuild the entire layers of minima data structure. We also recompute the score of each point and reconstruct the (a, b)-trees.

The following theorem analyzes the cost of the global rebuilding operation.

THEOREM 3. *The global rebuilding cost of the data structures is $O(n \log n)$ time and $O(n)$ space.*

PROOF. The result is derived from Theorem 2 and the discussion prior to it. Note that the dynamic range counting data structure is only built once (in the preprocessing phase) and it is not built again in any global rebuilding process applied to the data structures. □

We perform the global rebuilding step once in every $\sqrt{n}$ updates. An update up to the $(k+\sqrt{n})$-th layer, requires $O(\log^2 n + (k+\sqrt{n})k \log n)$ amortized time. We perform $\sqrt{n}$ such updates and then we globally rebuild the data structures in $O(n \log n)$ time so the amortized time for an update over $\sqrt{n}$ updates is $O(\sqrt{n} \log n + (k+\sqrt{n})k \log n) = O((k+\sqrt{n})k \log n)$.

The global rebuilding technique can also be applied on the method of Section 3.4 to obtain a data structure that handles insertions and deletions with reduced update cost. The results in this section are outlined in the following theorem:

THEOREM 4. *Given a set of n 2-dimensional points, we can support updates in $O((k+\sqrt{n})k \log n)$ amortized time and top-k dominating queries in $O(k)$ time. Alternatively, we are able to support insertions and deletions in $O((k+\sqrt{n}) \log n)$ amortized time and top-k dominating queries in $O(k \log n)$ time. Both data structures are constructed in $O(n \log n)$ time and use $O(n)$ space.*

5. RESULTS FOR WORD-RAM

In the previous results we have focused on the RAM model of computation. We can obtain slightly faster update algorithms for the semi-dynamic algorithm we presented by extending our results to the *word-RAM* model of computation. In the unit-cost *word-RAM model* [11], the memory is represented as an array of infinite cells (words) with each word storing w bits. The input elements are considered to be integers from the universe

$[U]^2 = \{0, \ldots, 2^w - 1\}^2$ so that any word can be addressed by any other word (through the use of a pointer).

The model supports random access of words as well as comparisons, arithmetic, shift and bitwise operations between words in constant time. In this work, we make the assumption that $w = \Theta(\log n)$ where n is the input's dataset size. This fact permits an input point or an index to the data structure to fit in a single word. The space cost under the word-RAM model is defined with respect to the number of words occupied; while the query and update times with respect to the number of word accesses and comparisons or operations needed to answer a query or perform an update respectively. The word-RAM model is a realistic model of computation, with integers of bounded precision, that closely emulates the mechanics of many programming languages (C, Python, Java, etc.).

To obtain our results, we use the dynamic range counting data structure of He and Munro [12] which, for word size $w = \Omega(\log n)$, supports queries in $O((\frac{\log n}{\log \log n})^2)$ worst-case time, insertions and deletions in $O((\frac{\log n}{\log \log n})^2)$ amortized time and uses $O(n)$ space.

The construction cost of our data structure is equal to the cost of constructing the dynamic range counting data structure, computing the score of each point, computing the layers of minima and constructing the (a, b)-trees. To build the dynamic range counting data structure we insert each point in the data structure for a total of $O(n(\frac{\log n}{\log \log n})^2)$ amortized time. The score of all points can be computed using the data structure in $O(n(\frac{\log n}{\log \log n})^2)$ total amortized time. The layers of minima can be built in $O(n \log n)$ time.

Lastly, using Lemma 4 the (a, b)-trees are built in $O(n \log k)$ time. Apart from the dynamic range counting data structure, the insertion and query algorithms remain the same. Combining the above observations we obtain the following result.

THEOREM 5. *Given a set of n 2-dimensional points in the word-RAM model with word size $w = \Theta(\log n)$, we can build a data structure that supports insertions of new points in $O(\frac{\log n}{\log \log n})^2 + k^2 \log n)$ amortized time and top-k dominating queries in $O(k)$ worst-case time. Alternatively, we can build a data structure that supports insertions of new points in $O(\frac{\log n}{\log \log n})^2 + k \log n)$ amortized time and top-k dominating queries in $O(k \log n)$ worst-case time. Both data structures are constructed in $O(n(\frac{\log n}{\log \log n})^2)$ amortized time and use $O(n)$ space.*

6. CONCLUSIONS AND FUTURE WORK

In this work, we have developed for the first time, algorithms for answering semi-dynamic and fully-dynamic top-k dominating queries in the 2-dimensional space, with non-trivial performance guarantees. In our solutions, k is a parameter that is considered fixed between queries.

The algorithms we have studied in this paper constitute the first attempt to process top-k dominating queries offering asymptotic performance guarantees for both their time and space cost. Existing work in the are is based completely on heuristic solutions built on top of access methods that work well in practice (e.g., R-trees).

Since object ranking in databases is a fundamental operation with many applications, we highlight some interesting research directions for future work in the area:

- An interesting and challenging problem is to lower the update cost for the fully-dynamic algorithms, by avoiding the global rebuilding technique.
- A second direction is to provide efficient top-k dominating query processing for any number of dimensions.
- A third direction is to design efficient algorithms for the external memory model. A baseline approach could be based on the successive computation of the k first layers of minima using iterative skyline computation. However, the goal is to offer more efficient algorithms with better performance bounds.
- Finally, it is worth investigating top-k dominating queries under the streaming model of computation, by offering approximate results as well as accuracy vs performance trade-offs.

7. REFERENCES

[1] H. Almuhimedi, S. Wilson, B. Liu, N. M. Sadeh, and A. Acquisti. Tweets are forever: a large-scale quantitative analysis of deleted tweets. In *CSCW*, pages 897–908, 2013.

[2] J. L. Bentley and J. B. Saxe. Decomposable searching problems i: Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[3] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[4] H. Blunck and J. Vahrenhold. In-place algorithms for computing (layers of) maxima. *Algorithmica*, 57(1):1–21, 2010.

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

[6] G. S. Brodal, G. Lagogiannis, and R. E. Tarjan. Strict fibonacci heaps. In *STOC*, pages 1177–1184, 2012.

[7] T. M. Chan and M. Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *SODA*, pages 161–173, 2010.

[8] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[10] G. N. Frederickson and D. B. Johnson. The complexity of selection and ranking in $X + Y$ and matrices with sorted columns. *Journal of Computer and System Sciences*, 24(2):197 – 208, 1982.

[11] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[12] M. He and J. I. Munro. Space efficient data structures for dynamic orthogonal range counting. In *WADS*, pages 500–511, 2011.

[13] M. Kontaki, A. N. Papadopoulos, and Y. Manolopoulos. Continuous top-k dominating queries. *IEEE Transactions on Knowledge and Data Engineering*, 24(5):840–853, 2012.

[14] X. Lian and L. Chen. Top-k dominating queries in uncertain databases. In *EDBT*, pages 660–671, 2009.

[15] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *Monographs in Theoretical Computer Science. (EATCS)*. Springer, 1984.

[16] M. H. Overmars and J. van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters*, 12(4):168–173, 1981.

[17] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Transactions on Database Systems*, 30(1):41–82, 2005.

[18] E. Tiakas, A. N. Papadopoulos, and Y. Manolopoulos. Progressive processing of subspace dominating queries. *The VLDB Journal*, 20(6):921–948, Dec. 2011.

[19] X. Xie, H. Lu, J. Chen, and S. Shang. Top-k neighborhood dominating query. In *DASFAA*, pages 131–145, 2013.

[20] M. L. Yiu and N. Mamoulis. Efficient processing of top-k dominating queries on multi-dimensional data. In *VLDB*, pages 483–494, 2007.

[21] M. L. Yiu and N. Mamoulis. Multi-dimensional top-k dominating queries. *The VLDB Journal*, 18(3):695–718, 2009.

[22] W. Zhang, X. Lin, Y. Zhang, J. Pei, and W. Wang. Threshold-based probabilistic top-k dominating queries. *The VLDB Journal*, 19(2):283–305, Apr. 2010.