Processing a Large Number of Continuous Preference Top-k Queries*

Albert Yu
Duke University
syu@cs.duke.edu

Pankaj K. Agarwal
Duke University
pankaj@cs.duke.edu

Jun Yang
Duke University
junyang@cs.duke.edu

ABSTRACT

Given a set of objects, each with multiple numeric attributes, a *(preference) top-k query* retrieves the k objects with the highest scores according to a user *preference*, defined as a linear combination of attribute values. We consider the problem of processing a large number of continuous top-k queries, each with its own preference. When objects or user preferences change, the query results must be updated. We present a dynamic index that supports the *reverse top-k query*, which is of independent interest. Combining this index with another one for top-k queries, we develop a scalable solution for processing many continuous top-k queries that exploits the clusteredness in user preferences. We also define an approximate version of the problem and present a solution significantly more efficient than the exact one with little loss in accuracy.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*

General Terms

Algorithms

Keywords

Linear preference, continuous top-k queries, reverse top-k queries

1 Introduction

In many applications, users are interested only in a small number (say, k) of "top" objects from a large set. If the objects have multiple numeric attributes, how to rank these objects depends on each user's preference, oftentimes specified as vector of weights that defines a linear combination of the attribute values. The weight associated with an attribute reflects the "importance" of that attribute to the user. For example, a real estate agency may list houses for sale with attributes such as listing price, year built, size of living area, lot size, etc. Each user is shown the highest ranked houses according his or her preference, i.e., those with the highest results for the linear combination. A user who cares most about the size of living area may assign the largest weight to this attribute (assuming that values of different attributes have been appropriately normalized relative to each other). On the other hand, a user who enjoys a yard more than indoor space may give the lot size a larger weight than the size of the living area. Because of the wide range of applications, there has been a lot of work on preference top-k queries [15, 17, 18, 19, 27].

Motivated by applications in business analysis, Vlachou et al. introduced the "reverse" top-k query [28]. In this setting, in addition to the set of objects of interest, we are given a set of user preferences. For a new object, we want to find which users would rank the new object in their top k; this information would allow a business analyst to assess, for example, the impact of a new product (object) on customers (users) relative to existing products.

Beyond the reverse top-k query, application settings such as data stream monitoring and publish/subscribe give rise to the problem of scalably processing a large number of *continuous* top-k queries [24], which can be thought of as a fully dynamic version of the reverse top-k query processing problem. In addition to handling changes to the set of user preferences, we need to maintain the top k objects under each user preference when objects are inserted, deleted, or updated. Consider again the example of real estate listing. Houses may come on and go off the market, and their information may be updated (such as lowering the listing price). We would like to notify all users of the changes (if any) to their top k houses. A new or updated house may make its way into some user's top-k list, while a deleted or updated listing may remove a house from a top-k list—in which case a replacement k-th ranked house must be added to the list. As another example, consider an investor who monitors the stock market in real time to identify profitable trades. The stocks will be ranked according to a wide range of numeric attributes, including, for example, trade volume and price change since market opening today, maximum swing during the last 30 minutes, price-to-earning ratio, average analyst rating, etc. Ranking preferences depends on whether the list identifies buying or selling opportunities, and will vary according to one's personal investing style and tolerance for risk. Each top-k list must be maintained as the market moves. In markets such as stocks, futures, and online auctions, both the volume of object updates and the number of preferences can be large, and the processing time requirement is demanding.

Despite much related work under various settings, e.g., [17, 24, 28], there still lacks a scalable, comprehensive solution to the problem of processing a large number of continuous top-k queries. Earlier results [17, 24, 28] rely heavily on heuristics, which have worked for the problem sizes they were intended for. However, they have linear query time or quadratic space in the worst case, unable to

*This work is supported by NSF grants CNS-05-40347, CCF-06-35000, IIS-07-13498, and CCF-09- 40671, by ARO grants W911NF-07-1-0376 and W911NF-08- 1-0452, by an NIH grant 1P50-GM-08183-01, and by a grant from the U.S.-Israel Binational Science Foundation.

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

handle dynamic updates efficiently, and are difficult to scale up further. For example, Mouratidis et al. [24] capped evaluation at 5,000 preferences; Vlachou et al. [28] tested up to 150,000 preferences, but the workloads did not include object updates, which are expensive under their approach. We aim to scale to a million preferences with both object and preference updates.

Approach and contributions. We approach the problem of processing a large number of continuous top-k queries with a geometric framework. Preference top-k queries are closely related to the concepts of *arrangement* and k-*level* [8] in discrete geometry, as previous work on ad hoc top-k queries by Das et al. [17] has identified. In this paper, we offer an intuitive interpretation of the k-level as a *query response surface (QRS)*, which geometrically represents the k-th ranked object over the space of all possible preference vectors. Within this framework, we apply three novel ideas in the setting of scalable continuous top-k query processing:

- ***Connection to halfspace range queries:*** We draw the connection between *halfspace range queries* [1, 6, 13, 16] and reverse top-k queries. This connection allows us to leverage results in computational geometry on halfspace range searching to devise an index for reverse top-k queries, which, in addition to being of independent interest, serves as a critical component of our solution to the scalable continuous top-k query processing problem.
- ***Combining preference- and QRS-driven processing:*** We sometimes need to evaluate multiple preference top-k queries simultaneously. Specifically, deleting an object may necessitate computing the new k-th ranked object for many preferences. A *preference-driven* approach runs these queries independently, which is suboptimal for clusters of preferences that share common top-k results. A *QRS-driven* approach identifies regions of the QRS within which top-k queries return the same k-th ranked object, and evaluates a single query for all preferences in each such region. However, the QRS, which depends only on the object distribution, can become very complex in high dimensions, with many regions containing few or no preferences at all. We propose a hybrid approach that combines the best of both approaches—using preference-driven processing for regions with few or no preferences, and using QRS-driven processing for dense clusters of preferences.
- ***Approximation:*** Not all applications require exact answers. By approximating QRS with a simpler surface, we reduce its complexity and, in turn, improve the efficiency of our algorithms. Specifically, we use the notion of *coresets*, which has been successfully used for geometric approximation algorithms [3, 4], to maintain a small subset of objects that induce a QRS closely approximating the QRS induced by the entire set of objects. Surprisingly, the size of the subset depends only on k and the approximation error, and not on the number of objects.

Our framework and ideas lead us to the following results:

- Leveraging the connection between reverse top-k and halfspace range queries, we obtain data structures for reverse top-k queries with linear space and sublinear query time in any fixed dimension. Experiments show orders-of-magnitude performance improvement and better scalability over the previous solution [28].
- We provide a scalable, comprehensive solution for processing a large number of continuous top-k queries. Our solution is fully dynamic in that it handles both object and preference updates efficiently. Experiments show that our hybrid approach achieves good performance by exploiting the clusteredness in user preferences while avoiding maintaining the full QRS.
- We define and solve a novel, approximate version of the problem. Experiments show that approximation significantly reduces processing costs with little loss in accuracy, allowing us to scale to even larger problem sizes.

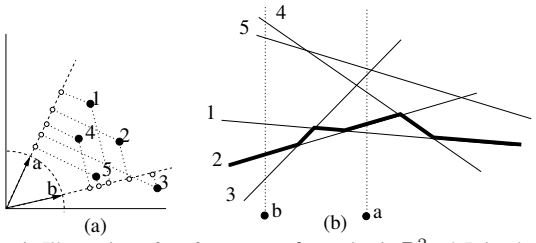


Figure 1. Illustration of preference top-k queries in $\mathbb{R}^2$. **a)** Primal setting: $\pi_{\leq 5}(a, \mathcal{O}) = \langle 1, 2, 4, 3, 5\rangle$; $\pi_{\leq 5}(b, \mathcal{O}) = \langle 3, 2, 1, 5, 4\rangle$. **b)** Dual setting: arrangement $\mathcal{A}(\mathcal{O}^*)$ and vertical lines a^* and b^*, where the thick polyline shows $\mathcal{A}_2(\mathcal{O}^*)$ (the 2-level).

In closing, we also note that our framework and solutions can apply to settings beyond those targeted in this paper, such as reverse nearest-neighbor queries, and preferences that are unknown or uncertain. We briefly comment on these applications in Section 7.

2 Preliminaries

2.1 Problem Statement

An *object* has d real-valued attributes and is represented as a point $(v_1, \ldots, v_d) \in \mathbb{R}^d$. A *preference* is represented as a unit vector, i.e., a point $(w_1, \ldots, w_d)$ on $\mathbb{S}^{d-1}$, the $(d-1)$-dimensional unit sphere embedded in $\mathbb{R}^d$. Each $w_i \geq 0$ is the *weight* for the i-th attribute. The *score* of an object o with respect to a preference q is $\langle q, o\rangle = \sum_{1\leq i\leq d} w_i v_i$. A hyperplane h normal to a preference vector q is of the form $\langle q, x\rangle = t$ for some $t \in \mathbb{R}$. All objects lying on h have the same score with respect to q, namely t.

Let $\mathcal{O} = \{o_1, o_2, \ldots, o_n\} \subset \mathbb{R}^d$ denote the set of n objects of interest. For simplicity, we assume that no two objects have the same score for any preference we consider. With a slight care, our framework and algorithms can be extended to handle ties. For a preference q, let $\pi_i(q, \mathcal{O})$ denote the *i-th ranked object* in $\mathcal{O}$ with respect to q; i.e., there are exactly $i-1$ objects $o' \in \mathcal{O}$ with $\langle q, o'\rangle < \langle q, o\rangle$. Let $\pi_{\leq i}(q, \mathcal{O}) = \{\pi_j(q, \mathcal{O}) \mid 1 \leq j \leq i\}$ denote the top i objects in $\mathcal{O}$ with respect to q. Geometrically, if we project the objects of $\mathcal{O}$ onto a line parallel to q, then $\pi_i(q, \mathcal{O})$ is the i-th farthest object on this line. Alternatively, if we sweep a hyperplane normal to q from $+\infty$ to $-\infty$, i.e., varying t from $+\infty$ to $-\infty$ for a hyperplane of the form $\langle q, x\rangle = t$, then $\pi_i(q, \mathcal{O})$ is the i-th object met by this hyperplane; see Figure 1(a). We are interested in the following:

- ***(Preference) top-k query:*** Given a query preference q, return $\pi_{\leq k}(q, \mathcal{O})$.
- ***Reverse (preference) top-k query:*** Given a set of m preferences $\mathcal{Q} = \{q_1, q_2, \ldots, q_m\}$ and a query object o, find the subset $\mathcal{Q}_o = \{q \in \mathcal{Q} \mid o \in \pi_{\leq k}(q, \mathcal{O} \cup \{o\})\}$, i.e., all preferences in $\mathcal{Q}$ for which o is one of the top-k objects.

In the fully dynamic version of the problem, called ***scalable continuous (preference) top-k query processing***,[1] given a set $\mathcal{O}$ of n objects and a set $\mathcal{Q}$ of m preferences, we want enable the maintenance of $\pi_{\leq k}(q, \mathcal{O})$ for all $q \in \mathcal{Q}$ at all times under the following operations.[2]

[1] "Scalable" highlights the emphasis on simultaneously processing a large number of preferences given as $\mathcal{Q}$. Contrast this problem to simply "continuous query processing," which considers one continuous query.

[2] In other words, a user with preference q will be able to maintain $\pi_{\leq k}(q, \mathcal{O})$ incrementally given the output computed by our solution for

- *Object insertion.* Given a new object o, find the subset $\mathcal{Q}_o = \{q \in \mathcal{Q} \mid o \in \pi_{\leq k}(q, \mathcal{O} \cup \{o\})\}$ and add o to $\mathcal{O}$.
- *Object deletion.* Given an object $o \in \mathcal{O}$ to be deleted, find the subset of preferences q such that $o \in \pi_{\leq k}(q, \mathcal{O})$, compute $\pi_k(q, \mathcal{O} \setminus \{o\})$ for each such q, and remove o from $\mathcal{O}$.
- *Preference insertion.* Add a preference q to $\mathcal{Q}$. Find $\pi_{\leq k}(q, \mathcal{O})$.
- *Preference deletion.* Remove a preference q from $\mathcal{Q}$.

Note that updates of existing objects and preferences can be modeled as deletions followed by insertions. It is not difficult to extend our framework and algorithms to handle updates directly, which would be more efficient than handling them as separate deletions and insertions. We omit the details because of space constraints.

In some cases, users do not need to know the exact top k objects, as long as they see a list sufficiently "close" to the exact one. In ***continuous approximate (preference) top-k query processing***, given a user-specified error tolerance $\varepsilon \in (0, 1)$, each user with preference q maintains a set $\tilde{\pi}_{\leq k}(q, \mathcal{O})$ of k objects such that, at all times, for all $o \in \tilde{\pi}_{\leq k}(q, \mathcal{O})$, $\langle q, o\rangle \geq \langle q, \pi_k(q, \mathcal{O})\rangle - \varepsilon \bar{d}(q, \mathcal{O})$, where $\bar{d}(q, \mathcal{O}) = \max_{o \in \mathcal{O}} \langle q, o\rangle - \min_{o \in \mathcal{O}} \langle q, o\rangle$ denotes the *extent* of the set of objects along the preference vector, i.e., the difference between the maximum and minimum scores.[3] Intuitively, all objects in approximate result are guaranteed to score higher than or not far from the actual k-th ranked object.

2.2 Geometric Framework and QRS

We first introduce a few geometric concepts as well as the notion of a *query response surface*, which will be useful to our algorithms.

Duality. The *duality transform* (see [23] for details) maps a point $p = (p_1, \ldots, p_d) \in \mathbb{R}^d$ to the hyperplane $p^* : x_d = p_1 x_1 + \ldots + p_{d-1} x_{d-1} - p_d$; and it maps a hyperplane $h : x_d = a_1 x_1 + \ldots + a_{d-1} x_{d-1} + a_d$ to the point $h^* = (a_1, \ldots, a_{d-1}, -a_d)$. It can be verified that the dual of p^* is p itself, i.e., $p^{**} = p$, and that if p lies above (resp. below, on) h, then h^* lies above (resp. below, on) p^*. For a unit vector $w \in \mathbb{S}^{d-1}$ with $w_d \neq 0$, the set of hyperplanes normal to w, i.e., of the form $\langle x, w\rangle = t$ where $t \in \mathbb{R}$, map to the vertical (x_d-axis parallel) line $w^* = \{(-w_1/w_d, \ldots, -w_{d-1}/w_d, t) \mid t \in \mathbb{R}\}$. We assume that w^* is oriented in $(+x_d)$-direction.

Let $\mathcal{O}^* = \{o_i^* \mid 1 \leq i \leq n\}$ be the set of hyperplanes dual to the objects in $\mathcal{O}$. Let $\mathcal{Q}^* = \{q_i^* \mid 1 \leq i \leq m\}$ be the set of vertical lines dual to the preferences in $\mathcal{Q}$. For a preference q, if $o = \pi_i(q, \mathcal{O})$, then o^* is the i-th hyperplane in $\mathcal{O}^*$ intersected by the $(+x_d)$-oriented line q^*. Hence, the first i hyperplanes of $\mathcal{O}^*$ intersected by q^* are dual to the objects in $\pi_{\leq i}(q, \mathcal{O})$; see Figure 1(b).

Arrangement. Let H be a set of r hyperplanes in $\mathbb{R}^d$. The *arrangement* of H, denoted by $\mathcal{A}(H)$, is the decomposition of $\mathbb{R}^d$ into *faces* induced by H, such that each face is the maximal connected region of $\mathbb{R}^d$ that lies in the same subset of H. $\mathcal{A}(H)$ is composed of $O(r^d)$ i-dimensional faces for $i = 0, \ldots, d$. See [8] for details. The *level* of a point p with respect to H, denoted by $\lambda(p, H)$, is the number of hyperplanes of H lying on or below p. Note that all points lying on the same face of $\mathcal{A}(H)$ have the same level. For $1 \leq k \leq |H|$, the *k-level* of $\mathcal{A}(H)$, denoted by $\mathcal{A}_k(H)$, is the closure of facets of $\mathcal{A}(H)$ whose level is k. $\mathcal{A}_k(H)$ is a piecewise-linear surface, and any line parallel to the x_d-axis intersects $\mathcal{A}_k(H)$ once; see Figure 1(b).

Query response surface. The following lemma establishes the connection between the concept of k-level and top-k queries.

Lemma 1. *For a preference q, if the intersection point of its dual line q^* with $\mathcal{A}_i(\mathcal{O}^*)$ lies on the hyperplane o^*, then $o = \pi_i(q, \mathcal{O})$.*

Therefore, we can view $\mathcal{A}_i(\mathcal{O}^*)$ as the *query response surface (QRS)* for the query returning the i-th ranked object under a user preference. Specifically, this QRS encodes, for any possible preference, the identity of the i-th ranked object. Each facet of this QRS corresponds to a set of preferences sharing a same i-th ranked object. Overall, $\bigcup_{1 \leq i \leq k} \mathcal{A}_i(\mathcal{O}^*)$ encodes $\pi_{\leq k}(q, \mathcal{O})$ for all possible preference vectors q.

2.3 Query Primitives

We describe two primitives that our algorithms will use repeatedly.

Halfspace range query. The problem is to preprocess a set P of n points in $\mathbb{R}^d$ so that all points in P lying above a query hyperplane h can be reported quickly. In dual, this problem corresponds to reporting all hyperplanes of P^* lying below the point h^*. Several approaches have been proposed for this query. For $d \leq 3$, a query can be answered in $O(\log n + t)$ time, where t is the output size, using $O(n)$ space [16, 1]. For $d \geq 4$, given a parameter $n \leq s \leq n^{\lceil d/2 \rceil}$, a query can be answered in $O((n/s^{1/\lceil d/2 \rceil}) \log n + t)$ time using $O(s^{1+\varepsilon})$ space for any $\varepsilon > 0$ [22]. The known lower bounds [12] suggest that these bounds are close to optimal. I/O-efficient indexing schemes for halfspace range queries were given in [2]; dynamic schemes were presented in [6, 13]; see also [15].

Since the focus of this paper is not to develop the best possible index for halfspace range searching, our experiments in Section 5 simply use a tree index on P based on some hierarchical spatial partitioning scheme, such as a quad-tree or kd-tree, and answer halfspace range queries in a straightforward top-down manner.[4] This index offers adequate performance in our experiments, but it can certainly be replaced by more sophisticated index with better worst-case guarantees, without affecting the rest of our solution. We will assume that a halfspace range query can be answered in $O(\mathsf{q}(n) + t)$ time, and a point can be inserted or deleted in $O(\mathsf{u}(n))$ time.

Top-k query. There is a close relationship between halfspace range queries and top-k queries. Indeed, let q be a query preference for which we wish to report $\pi_{\leq k}(q, \mathcal{O})$. Let h be a hyperplane normal to q of the form $\langle q, x\rangle = t$, where $t \in \mathbb{R}$. We perform a halfspace range query over $\mathcal{O}$ with respect to h. If it returns fewer than k objects, we decrease the value of t and try again. If it attempts to report more than k points, we stop, increase t, and then try again. Thus, by doing a binary search, we can find a value of t such that exactly k objects are reported. This procedure takes $O((\mathsf{q}(n) + k) \log n)$ time. Using the index of [22], the running time can be improved to $O(\mathsf{q}(n) + k)$. Conversely, an index for top-k queries, in which a user can specify the value of k as part of the query, can be adapted to answer halfspace range queries. We will thus use $O(\mathsf{q}(n) + k)$ to denote the query time for a top-k index and $O(\mathsf{u}(n))$ to denote the update time. In our implementation, we simply use a quad-tree or kd-tree to answer top-k queries with a branch-and-bound method. It can be easily replaced by a more sophisticated one without affecting the rest of our solution.

Note that we are sometimes interested only in the k-th ranked object (instead of all top k objects). When k is small (i.e., $k \leq$

the following operations. In fact, our solution by itself does not need to store this list for every preference.

[3]We use $\varepsilon \bar{d}(q, \mathcal{O})$ instead of $\varepsilon \langle q, \pi_1(q, O)\rangle$ as the error measure because the former is independent of the choice of origin and is smaller than the latter if all object attributes have non-negative values. See the last remark in Section 4.1 for more discussion on an alternative formulation.

[4]Given a query hyperplane h, we search the tree top-down as follows. At a node v with bounding box B_v, if B_v lies completely below h, we do nothing; otherwise, we recursively search all children of v, or, if v is leaf, return all points indexed by v that lie above h.

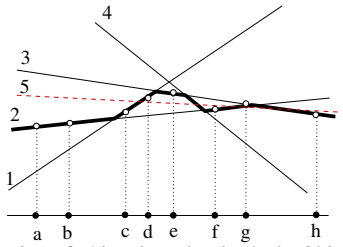


Figure 2. Illustration of object insertion in dual. Objects $(1, \ldots, 4)$ are shown as solid lines and preferences $(a, b, \ldots, h)$ are shown as vertical lines. Prior to inserting object 5 (shown as a hashed line), the 2-level is shown as the thick polyline, and the cutoff points are shown as white dots. Insertion of 5 changes the top-2 lists for preferences d, e, and g, whose cutoff points lie above the newly inserted dual line.

$\mathsf{q}(n)$), simply running a top-k query and returning only the k-th object works well.

2.4 Summary of Our Results

First, we show that a reverse top-k query can be formulated as a halfspace range query and thus can be answered in $O(\mathsf{q}(m) + t)$ time, where m is the number of preferences and t is the number of them affected by the query object. As far as we know, this is the first linear-size index that can answer this query in sublinear time in any fixed dimension. For $d \leq 3$, the query time is $O(\log m + k)$, which is optimal. Experiments show our approach to be much faster than the current state of the art [28, 29].

Second, we present a scalable solution for processing many continuous top-k queries, which maintains $\pi_{\leq k}(q, \mathcal{O})$ for a set $\mathcal{Q}$ of preferences under both object and preference updates. We start by outlining two approaches—preference-driven and QRS-driven—for finding the new k-th ranked object for each preference affected by an object update. The preference-driven approach evaluates one such query for each affected preference; the QRS-driven approach evaluates one query for each facet of $\mathcal{A}_k(\mathcal{O}^*)$ (within which all preferences have the same k-th ranked object). We adopt a hybrid approach that uses the preference-driven one for sparse preferences and the QRS-driven one for clustered preferences. Experiments show that this hybrid approach achieves good performance by exploiting the clusteredness of preferences while avoiding maintaining complex regions of the QRS with sparse preferences.

Third, we show that if approximate answers as described in Section 2.1 are acceptable, we can compute a subset $\mathcal{C} \subseteq \mathcal{O}$ of size $O(k/\varepsilon^{(d-1/2)})$, such that $\langle q, \pi_{\leq k}(q, \mathcal{C})\rangle \geq \langle q, \pi_k(q, \mathcal{O})\rangle - \varepsilon\bar{d}(q, \mathcal{O})$ for all preference q. The set $\mathcal{C}$ can be maintained efficiently under insertion and deletion of objects. Experiments show that this approach significantly reduces the complexity of QRS and improves running time with little loss of accuracy.

Finally, our results have a number of applications beyond those focused on by this paper; we discuss them in Section 7.

3 From Reverse Top-k to Continuous Top-k Queries

We now present our solutions for answering reverse top-k queries and for processing a large number of continuous top-k queries. We start with a static solution for reverse top-k queries, ignoring object or preference updates. We then describe a fully dynamic solution that handles both object and preference updates.

3.1 A Static Solution for Reverse Top-k

Given a set $\mathcal{O}$ of objects, a set $\mathcal{Q}$ of preferences, and a query object $o \notin \mathcal{O}$, we wish to report the subset of preferences $\mathcal{Q}_o = \{q \in \mathcal{Q} \mid o \in \pi_{\leq k}(q, \mathcal{O} \cup \{o\})\}$. Intuitively, a preference q can be affected by o only if o scores higher than the current k-th ranked object for q. In dual, this intuition translates into the following lemma, which characterizes the set $\mathcal{Q}_o$.

Lemma 2. *For a query object o, $q \in \mathcal{Q}_o$ iff $q^* \cap \mathcal{A}_k(\mathcal{O}^*)$ lies above the dual hyperplane o^*.*

Proof. Let $p = \pi_k(q, \mathcal{O})$. By Lemma 1, $q^* \cap \mathcal{A}_k(\mathcal{O}^*) = q^* \cap p^*$. If the new object o belongs to $\pi_{\leq k}(q, \mathcal{O} \cup \{o\})$, then $p = \pi_{k+1}(q, \mathcal{O} \cup \{o\})$. By Lemma 1, q^* intersects o^* before intersecting p^*, implying that $q^* \cap \mathcal{A}_k(\mathcal{O}^*)$ lies above o^*; see Figure 2. □

In view of Lemma 2, we index, in the dual space, the point set $\hat{\mathcal{Q}} = \{q^* \cap \mathcal{A}_k(\mathcal{O}^*) \mid q \in \mathcal{Q}\}$, i.e., intersection points between the vertical lines in $\mathcal{Q}^*$ and the k-level. We call these points the *cutoff* points. To create this index, we compute $\hat{\mathcal{Q}}$ by performing a top-k query on $\mathcal{O}$ with each preference q to find $\pi_k(q, \mathcal{O})$, using an index on $\mathcal{O}$ supporting top-k queries as described in Section 2.3.[5] We then preprocess $\hat{\mathcal{Q}}$ into an index for halfspace range queries so that we can report all points of $\hat{\mathcal{Q}}$ lying above a query hyperplane. By Lemma 2, a reverse top-k query can be answered in $O(\mathsf{q}(m) + t)$ time, where t is the output size.

Using the results in [1, 22], discussed earlier in Section 2.3, we arrive at the following result for answering reverse top-k queries. As noted in Section 2.3, simpler, more practical methods can be used instead, but with weaker theoretical bounds.

Theorem 1. *Let $\mathcal{O}$ be a set of n objects in $\mathbb{R}^d$ and $\mathcal{Q}$ a set of m preferences. 1) For $d \leq 3$, $\mathcal{Q}$ can be preprocessed in $O((n + m)\log n)$ time into an index of size $O(m)$ so that a reverse top-k query can be answered in $O(\log m + t)$ time, where t is the output size. 2) For $d \geq 4$ and for a parameter $m \leq s \leq m^{\lceil d/2\rceil}$, there is an index of size $O(s^{1+\varepsilon})$ for any $\varepsilon > 0$, so that a reverse top-k query can be answered in $O((m/s^{1/\lceil d/2\rceil})\log m + t)$ time, where t is the output size.*

We note that the above solution allows each user (preference) to choose a different value of k; the cutoff point to be indexed for each user would be defined by the value of k specific to the user.

3.2 A Fully Dynamic Solution

Building on the static solution for reverse top-k queries, we now show how to process a large number of continuous top-k queries in a fully dynamic setting, with both object and preference updates. We discuss three approaches, distinguished primarily by their handling of preferences affected by object updates. We start by outlining two possible approaches with complementing strengths (and weaknesses), and then describe the approach of our choice—a hybrid combining the advantages of the first two approaches. All three approaches employ an index on the set of n objects $\mathcal{O}$, which supports preference top-k queries in $O(\mathsf{q}(n) + k)$ time and object insertions and deletions in $O(\mathsf{u}(n))$ time, as discussed in Section 2.3.

3.2.1 Preference-Driven Approach

In addition to the index on $\mathcal{O}$ for top-k queries, this approach employs an index on the set $\hat{\mathcal{Q}}$ of cutoff points discussed in Section 3.1. Inserting a preference q involves a top-k query against the index on $\mathcal{O}$ to initialize q's list of top k objects. We then compute the cutoff point for q from its k-th ranked object, and insert it into the index on $\hat{\mathcal{Q}}$. Deleting a preference q simply entails deleting its cutoff point from the index on $\hat{\mathcal{Q}}$. Thus, the insertion and deletion times are $O(\mathsf{u}(m) + \mathsf{q}(n) + k)$ and $O(\mathsf{u}(m))$, respectively.

[5] Instead of performing each top-k query individually, we can batch them using, for example, the QRS-driven or hybrid approach in Section 3.2.

Now consider the insertion (or deletion) of an object o. First, we issue a halfspace range query with o^* against the index on $\hat{\mathcal{Q}}$ to find the set of affected preferences $\mathcal{Q}_o \subseteq \mathcal{Q}$, which correspond to the cutoff points lying above (or, for deletion of o, on or above) o^*, as in Section 3.1. We also update the index on $\mathcal{O}$ with o. Next, for each affected preference $q \in \mathcal{Q}_o$, we issue a top-k query with q against the index on $\mathcal{O}$ to find the new k-th ranked object p, and then update the index on $\hat{\mathcal{Q}}$ with the new cutoff point for q, given by $q^* \cap p^*$. The list of top k objects for q can be easily maintained using o (or, for deletion of o, o and p). The total update time is $O(\mathsf{q}(m) + \mathsf{u}(n) + t(\mathsf{q}(n) + \mathsf{u}(m)))$, where t is the number of affected preferences.

We call this approach *preference-driven* because it issues a separate top-k query for each affected subscription in $\mathcal{Q}_o$, which can be expensive if $\mathcal{Q}_o$ is large, and wasteful if many preferences share the same k-th ranked object. Intuitively, for "nearby" preferences with the same k-th ranked object, we would like to use only one query, which leads us to the next approach.

3.2.2 QRS-Driven Approach

An alternative approach will be to leverage the query response surface $\mathcal{A}_k(\mathcal{O}^*)$. Recall from Section 2.2 that each facet of this QRS corresponds to a set of preferences sharing the same k-th ranked object, giving us a natural way to process preferences in groups.

To this end, in addition to the index on $\mathcal{O}$ for top-k queries, the *QRS-driven* approach maintains an index for $\mathcal{A}_k(\mathcal{O}^*)$. If an object o is inserted (or deleted), we update the index on $\mathcal{O}$ as well as the index for $\mathcal{A}_k(\mathcal{O}^*)$, querying the index on $\mathcal{O}$ as needed. The complexity of this operation does not depend on the number of preferences. Finally, for each new facet ϕ on the updated QRS, let p denote the object whose dual hyperplane p^* contains ϕ, and let $\mathcal{Q}_\phi$ denote the set of preferences q whose dual lines q^* intersect ϕ.[6] All preferences in $\mathcal{Q}_\phi$ have p as their new k-th ranked object, and their lists of top k objects can be maintained using o (resp. o and p).

Note that updating of the QRS is oblivious to the actual set of preferences. Indeed, the QRS-driven approach effectively computes, without any knowledge of $\mathcal{Q}$, a description (based on facets of $\mathcal{A}_k(\mathcal{O}^*)$) of the set of affected preferences, together with the incremental changes to their lists of top-k objects. This feature makes the QRS-driven approach attractive for some applications (such as *monochromatic reverse top-k* queries in business analysis [28]), a point we shall come back to in Section 7.

There are two difficulties with this approach, however. First, the QRS can be large and complex to update, especially in higher dimensions. Second, many parts of the QRS may have few or no preferences, so it would be a waste of effort to maintain the QRS for these parts. These observations lead us to the idea of combining this approach with the preference-driven approach earlier.

3.2.3 Hybrid Approach

For the preference-driven approach, the index on $\hat{\mathcal{Q}}$ can be updated efficiently, but independently computing the new k-th ranked object for each affected preference can be inefficient. For the QRS-driven approach, identically affected preferences are processed efficiently as a group, but maintaining parts of the QRS with few or no actual preferences is wasteful. To get the best from both approaches, we adopt a hybrid approach that intelligently switches between the two processing modes.

In addition to the index on $\mathcal{O}$ for top-k queries, the hybrid approach maintains a search tree $\mathcal{T}$ based on a hierarchical spatial partitioning of the dual space (we use a quad-tree in our implementation). Each node v of $\mathcal{T}$ is associated with a bounding box $B_v \subseteq \mathbb{R}^d$. Let $\mathcal{Q}_v^* \subseteq \mathcal{Q}^*$ denote the set of vertical lines in $\mathcal{Q}^*$ stabbing B_v, and let $\mathcal{O}_v^* \subseteq \mathcal{O}^*$ denote the set of hyperplanes in $\mathcal{O}^*$ intersecting B_v. We store the following three counters at each node v: $m_v = |\mathcal{Q}_v^*|$, $n_v = |\mathcal{O}_v^*|$, and b_v^Δ, the number of hyperplanes in $\mathcal{O}_{p(v)}^*$ that lie below B_v, where $p(v)$ is the parent of v. The number of hyperplanes in $\mathcal{O}^*$ lying below B_v, denoted b_v, can be computed by summing b_u^Δ over each node u on the path from the root to v. At each leaf v of $\mathcal{T}$, we also store the sets $\mathcal{O}_v^*$ and $\mathcal{Q}_v^*$.

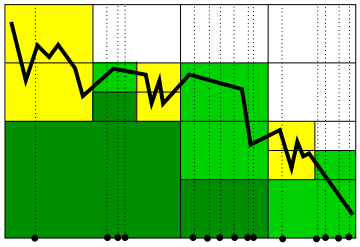


Figure 3. Leaves of $\mathcal{T}$. The QRS is shown as a thick polyline, and the dual lines of preferences are shown as dotted lines. Solid lines show the partitioning of the dual space into leaves; for clarity here we use equi-distance partitioning, though in practice it need not be the case. Black leaves are filled with a dark (green) shade; grey-dense leaves are filled with a medium (green) shade; grey-sparse leaves are filled with a light (yellow) shade; white leaves are not shaded.

A leaf v can be one of the following types (where τ_m and τ_n are user-defined parameters):

- *White* if $b_v > k$; i.e., B_v is strictly above $\mathcal{A}_k(\mathcal{O}^*)$.
- *Black* if $b_v + n_v < k$; i.e., B_v is strictly below $\mathcal{A}_k(\mathcal{O}^*)$.
- *Grey-sparse* if $(b_v \le k \le b_v + n_v) \wedge (m_v < \tau_m)$; i.e., B_v intersects $\mathcal{A}_k(\mathcal{O}^*)$ and contains few cutoff points.
- *Grey-dense* if $(b_v \le k \le b_v + n_v) \wedge (m_v \ge \tau_m) \wedge (n_v < \tau_n)$; i.e., B_v intersects $\mathcal{A}_k(\mathcal{O}^*)$ and likely contains many cutoff points, and $\mathcal{A}_k(\mathcal{O}^*)$ is not very complex.

These leaf types are depicted in Figure 3. A node v satisfying none of the conditions above passes the following *splitting condition*:

$$(b_v \le k \le b_v + n_v) \wedge (m_v \ge \tau_m) \wedge (n_v \ge \tau_n).$$

In this case, v is an interior node. Practically, we choose τ_m and τ_n to reflect 1) the "tipping point" when one of the preference- and QRS-driven approaches becomes more efficient than the other, and 2) the granularity at which we make such a decision.

Constructing $\mathcal{T}$. We start with a tree containing a single root node to consider, with $B_{\text{root}} = \mathbb{R}^d$, $m_{\text{root}} = m$, $n_{\text{root}} = n$, and $b_{\text{root}}^\Delta = b_{\text{root}} = 0$. At each node v we consider, we test the splitting condition. If v passes the splitting condition, we make v a non-leaf, partition B_v among its children, and consider each child. Otherwise, v is a leaf: we store $\mathcal{Q}_v^*$, $\mathcal{O}_v^*$, and determine v's type.

Object insertion. For the insertion of a new object o, we first update $\mathcal{T}$ top-down. At a node v, we increment n_v if o^* intersects B_v, or increment b_v^Δ if o^* lies below B_v. There are three cases:

1. *v was a non-leaf.* If now $b_v > k$, we contract the subtree rooted at v into a single white leaf and stop. Otherwise, v remains a non-leaf and we repeat the same procedure for each child of v, but skipping any child u where o^* lies above B_u.
2. *v was a white or black leaf.* The only case requiring action is when a previously black v becomes grey or non-leaf because now $b_v + n_v = k$. In this case, we construct a subtree rooted at v for $\mathcal{O}_v^*, \mathcal{Q}_v^*$ using the construction procedure above.
3. *v was a grey leaf.* The only case requiring action is when a previously grey-dense v turns into a non-leaf because n_v now reaches τ_n. In that case, we use the construction procedure to build a subtree rooted at v.

[6] This set can be either explicitly maintained for each facet of the QRS, or computed by searching another data structure on $\mathcal{Q}$.

After updating $\mathcal{T}$, we traverse $\mathcal{T}$ to compute, for each grey leaf v, the set of affected preferences in $\mathcal{Q}_v^*$:

- If v is grey-dense, the QRS inside B_v must be simple because few dual hyperplanes intersect B_v, so we take a QRS-driven approach. We compute the new facets of the QRS inside B_v (i.e., $\mathcal{A}_{k-b_v}(\mathcal{O}_v^*) \cap B_v$). For each new facet ϕ, let p denote the object whose dual hyperplane p^* contains ϕ. All preferences whose dual lines intersect ϕ have p as the new k-th ranked object.
- If v is grey-sparse, $\mathcal{Q}_v^*$ is small, so we take a preference-driven approach, issuing one top-k query against the index on $\mathcal{O}$ for each preference in $\mathcal{Q}_v^*$. If $\mathcal{O}_v^*$ happens to be small too, instead of using the index on $\mathcal{O}$, we can simply compute the $(k - b_v)$-th ranked object in $\mathcal{O}_v^*$ for each preference by scanning $\mathcal{O}_v^*$.

Object deletion. For the deletion of object o, we again first update $\mathcal{T}$ top-down. At a node v, we decrement n_v if o^* intersects B_v, or decrement b_v^Δ if o^* lies below B_v. There are three cases:

1. *v was a non-leaf.* If $b_v + n_v$ now drops below k, we contract the subtree rooted at v into a single black leaf. If n_v drops below τ_n but still $b_v + n_v \geq k$, we contract the subtree rooted at v into a single grey-dense leaf. Otherwise, v remains a non-leaf and we repeat the same procedure for each child of v, but skipping any child u where o^* lies above B_u.
2. *v was a white or black leaf.* The only case requiring action is when a previously white v becomes grey or non-leaf because now $b_v = k$. In this case, we construct a subtree rooted at v using the construction procedure above.
3. *v was a grey leaf.* We make v black if $b_v + n_v < k$.

After $\mathcal{T}$ has been updated, we compute the set of affected preferences and their new k-th ranked objects as discussed in the case of object insertion, switching between QRS- and preference-driven approaches as appropriate.

Preference insertion. For the insertion of a new preference q, we issue a top-k query with q against the index on $\mathcal{O}$ to find $\mathcal{O}_q = \pi_{\leq k}(q, \mathcal{O})$. Using $\mathcal{O}_q$ we calculate q's cutoff point $\hat{q}$, and search $\mathcal{T}$ for the grey leaf v such that $\hat{q} \in B_v$. For every node u along the path from the root to v, we increment m_u. If v was grey-sparse and now $m_v = \tau_m$, v would become grey-dense or non-leaf; in this case, we construct a subtree rooted at v using the construction procedure above.

Preference deletion. For the deletion of preference q, we search $\mathcal{T}$ for the grey leaf v such that B_v contains the cutoff point of q. For every node u along the path from the root to v, we decrement m_u. If (and as soon as) m_u drops from τ_m to $\tau_m - 1$ for any u we encounter during the search, we replace the subtree rooted at u with a single grey-sparse leaf.

4 Approximate Top-k Queries

As mentioned in Section 1, it suffices for users of many applications to have approximate lists of top-k objects under their preferences. In this section, we show that in this case we can build the index on a small subset of $\mathcal{O}$, and update the lists more efficiently. In Section 4.1, we show that such a small subset $\mathcal{C}$, called *coreset*, can be computed efficiently. Section 4.2 describes how to update the coreset efficiently as $\mathcal{O}$ changes. Section 4.3 further describes procedures for maintaining indexes based on $\mathcal{C}$ as well as the top-k lists of all users. As we will see, maintaining them upon every change to $\mathcal{C}$ is unnecessary and expensive—insertion or deletion of a single object in $\mathcal{O}$ sometimes causes multiple changes to $\mathcal{C}$. We therefore design these procedures to perform maintenance lazily only when necessary.

4.1 Computing a Coreset

For a unit vector $q \in \mathbb{S}^{d-1}$, the *extent* of $\mathcal{O}$ in direction q, denoted by $\bar{d}(q, \mathcal{O})$, is

$$\bar{d}(q, \mathcal{O}) = \max_{o \in \mathcal{O}} \langle q, o \rangle - \min_{o \in \mathcal{O}} \langle q, o \rangle,$$

i.e., the difference between the maximum and the minimum scores for the preference q. Given an integer $k \geq 1$ and a parameter $\varepsilon > 0$, a subset $\mathcal{C} \subseteq \mathcal{O}$ is called a (k, ε)-*coreset* (or simply *coreset* for brevity) if for all $i \leq k$ and $q \in \mathbb{S}^{d-1}$,

$$\langle q, \pi_i(q, \mathcal{C}) \rangle \geq \langle q, \pi_i(q, \mathcal{O}) \rangle - \varepsilon \bar{d}(q, \mathcal{O}). \tag{1}$$

We show that a coreset of size $O(k/\varepsilon^{(d-1/2)})$ can be computed efficiently.

Before describing the algorithm, we need a property of coreset, which will be critical for the algorithm. A linear transform $\Gamma : \mathbb{R}^d \to \mathbb{R}^d$ is called an *affine* transform if the matrix Γ is nonsingular—it includes translation, rotation, and scaling.

Lemma 3. *Let $k > 0$ be an integer, $\varepsilon > 0$ a parameter, and Γ an affine transform. A subset $\mathcal{C} \subseteq \mathcal{O}$ is a (k, ε)-coreset of $\mathcal{O}$ if and only if $\Gamma(\mathcal{C})$ is a (k, ε)-coreset of $\Gamma(\mathcal{O})$.*

The proof of this lemma, a slight variant of the one given in [30], is omitted here because of lack of space.

Converting $\mathcal{O}$ into a fat point set. For a constant $\alpha > 0$, $\mathcal{O}$ is called α-*fat* if

$$\max_{q_1, q_2 \in \mathbb{S}^{d-1}} \bar{d}(q_1, \mathcal{O}) / \bar{d}(q_2, \mathcal{O}) \leq \alpha.$$

We can compute an affine transform Γ such that $\Gamma(o)$ is α_d-fat for some constant α_d that depends on d. To this end, we first compute the approximate minimum-volume bounding box B for $\mathcal{O}$ with the algorithm of Barequet and Har-Peled [10], as follows. We pick the set $\mathcal{A}$ of d *anchor* objects $a_0, \ldots, a_d$, one by one. We choose a_0 arbitrarily, and we choose a_{i+1} to be the farthest object from $\mathrm{span}(a_0, \ldots, a_i)$, i.e., the span of all previously chosen anchors. The set $\mathcal{A}$ of anchor objects defines the bounding box B: a_0 lies in the center of B; the vector from a_{i+1} to $\mathrm{span}(a_0, \ldots, a_i)$ gives an direction orthogonal to the directions defined by $\{a_0, \ldots, a_i\}$ (see Figure 4(b)). Next, we compute a transform Γ, such that $\Gamma(B)$ maps to $[-1, +1]^d$. It can be checked that $\Gamma(\mathcal{O})$ is α_d-fat for a constant α_d (see, e.g., [3]).

Constructing $\mathcal{C}$. Given $\mathcal{O}$, we first compute the affine transform Γ, as described above, so that $\Gamma(\mathcal{O})$ is fat and $\Gamma(\mathcal{O}) \subset [-1, +1]^d$. Let S be the sphere of radius $\sqrt{d} + 1$ centered at the origin in $\mathbb{R}^d$. We construct a set $\mathcal{G}$ of *grid points* on S as follows. Set parameter $\delta = \beta\sqrt{\varepsilon}$ for a sufficiently small constant $0 < \beta < 1$. We choose a set $\mathcal{G} \subset S$ of $O(1/\beta^{d-1}) = O(1/\varepsilon^{(d-1)/2})$ points, so that for any point $x \in S$ there is a grid point $p \in \mathcal{G}$ such that $\|x - p\| \leq \delta$.

Next, for each grid point $p \in \mathcal{G}$, we compute $\sigma_k(p)$, the $(\varepsilon/2)$-approximate k nearest neighbors of p in $\Gamma(\mathcal{O})$; see Figure 4(c). We set $\mathcal{C} = \bigcup_{p \in \mathcal{G}} \sigma_k(p)$. By adapting the methods for answering approximate nearest-neighbor queries [9], the $(\varepsilon/2)$-approximate k nearest neighbors of a query point can be computed in $O(\log n + k/\varepsilon^d)$ time. In practice, we can use a branch-and-bound algorithm similar to the one used for answering a top-k query.

Theorem 2. *Given a set $\mathcal{O}$ of n objects, an integer $k > 0$, and a parameter $\varepsilon > 0$, a (k, ε)-coreset of $\mathcal{O}$ of size $O(k/\varepsilon^{(d-1)/2})$ can be computed in $O(n \log n + k/\varepsilon^{3d/2})$ time.*

Proof. Since $|\mathcal{G}| = O(1/\varepsilon^{(d-1)/2})$, $|\mathcal{C}| = O(k/\varepsilon^{(d-1)/2})$; the running time of the algorithm follows from the query time of the

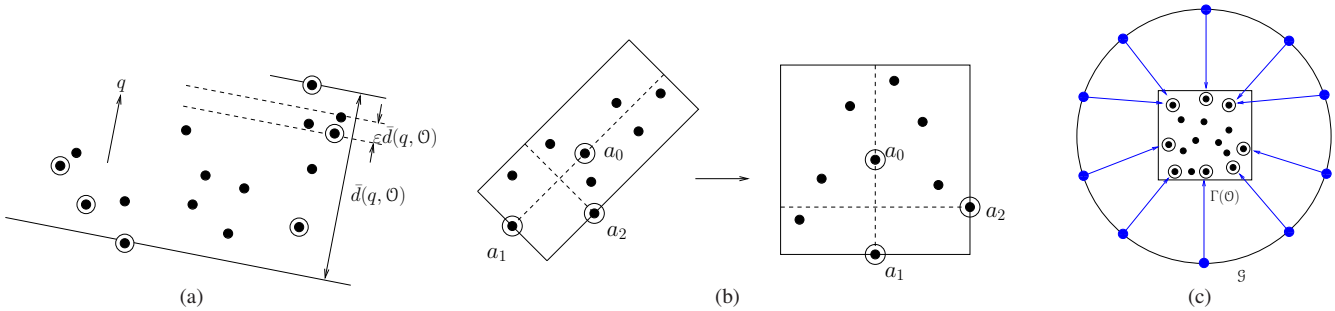


Figure 4. a) Illustration of coreset for $k = 2$. Points are shown as black dots and members of the coreset are circled. **b)** Converting $\mathcal{O}$ into a fat point set using an affine transform defined using anchor points $\{a_0, a_1, a_2\}$. **c)** Constructing the coreset by finding the k nearest neighbors in $\Gamma(\mathcal{O})$ (showing in the bounding box) of each grid point in $\mathcal{G}$ (shown on the sphere).

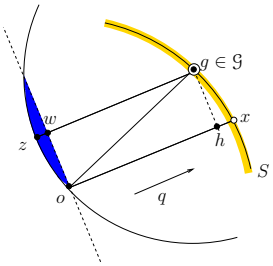


Figure 5. Correctness of the coreset construction algorithm.

approximate k-nearest neighbor data structure. It thus suffices to prove that $\mathcal{C}$ is a (k, ε)-coreset.

For each $q \in \mathbb{S}^{d-1}$ and for all $i \leq k$, we show that (1) holds. We prove this claim by induction on i. Suppose this claim holds for up to $i - 1$. Suppose $o = \pi_i(q, \mathcal{O})$. Let $x \in S$ be the intersection point of S with the ray emanating from o in direction q. Refer to Figure 5. Let $g \in \mathcal{G}$ be the closest grid point to $x \in S$. If o is the i-th nearest neighbor of g, then o is included in $\mathcal{C}$. Otherwise, the i-th nearest neighbor must lie in the shaded (blue) region. The error is within $\|w - z\|$, which can be shown to be less than $\|h - x\| < (\varepsilon/2)\bar{d}(q, \mathcal{O})$, provided that β is chosen sufficiently small; see [30]. Recall that we computed the $(\varepsilon/2)$-approximate k-nearest neighbors of g, so we can argue that if $\tilde{o}$ is the $(\varepsilon/2)$-approximate i-th nearest neighbor of g, then $\langle q, \tilde{o}\rangle \geq \langle q, o\rangle - \varepsilon\bar{d}(q, \mathcal{O})$. □

Remarks. Note that $\mathcal{C}$ approximates $\pi_{\leq k}(q, \mathcal{O})$ for all $q \in \mathbb{S}^{d-1}$. If we are interested in preferences $q = (q_1, \ldots, q_d)$ for which $q_i \geq 0$ for all $i \leq d$, then we choose only those points of $\mathcal{G}$ that are within or not far from the first orthant (we use those with coordinates no less than -0.3). The asymptotic bound on the size of $\mathcal{C}$ does not change, but the constant changes.

Agarwal et al. [5] define a coreset using a stronger definition of approximation which ensures that

$$\langle q, \pi_i(q, \mathcal{C})\rangle \geq (1 - \varepsilon)\langle q, \pi_i(q, \mathcal{O})\rangle \tag{2}$$

for all $1 \leq i \leq k$ and for all directions $q \in \mathbb{S}^{d-1}$ if all attributes are non-negative. Using a similar algorithm they show that a coreset of size $O(k/\varepsilon^{(d-1)/2})$ can be computed under this stronger definition. A result in [7] shows that the coreset can be maintained efficiently under insertion and deletion of objects. Our definition provides a weaker theoretical guarantee because the error is bounded in terms of extent, which depends on the position of highest ranked object. In particular, if the score of $\pi_k(q, \mathcal{O})$ is much smaller than $\pi_1(q, \mathcal{O})$, then the bound in (1) could be large. However, we choose our definition because in practice it also produces a very good approximation of $\pi_{\leq k}(q, \mathcal{O})$ for every q, and updating $\mathcal{C}$ under insertion or deletion of an object is considerably simpler.

4.2 Updating the Coreset

We discuss how to maintain the coreset $\mathcal{C}$ under insertion and deletion of objects. We maintain the set of anchor points $\mathcal{A}$, the bounding box B, and the affine transform Γ. To help reduce the amortized cost of reconstructing the coreset, we maintain the coreset as the union of two sets, i.e., $\mathcal{C} = \mathcal{C}^{\text{in}} \cup \mathcal{C}^{\text{out}}$, where $\mathcal{C}^{\text{out}}$ is used to "buffer" new objects that would otherwise trigger coreset reconstruction immediately; details now follow.

Object insertion. Suppose the new object o is inside the bounding box B. For each grid point $g \in \mathcal{G}$, if $\Gamma(o)$ is one of g's new (approximate) k nearest neighbors among $\Gamma(\mathcal{O} \cup \{o\})$, we insert o into $\mathcal{C}^{\text{in}}$ and remove the old k-th nearest neighbor of g from $\mathcal{C}^{\text{in}}$. Overall, $\mathcal{C}$ does not change unless o becomes one of the k nearest neighbors of some grid point, in which case o is inserted to $\mathcal{C}^{\text{in}}$ and one or more objects are deleted from $\mathcal{C}^{\text{in}}$.

If the new object o is outside B, the naive approach would be to reconstruct $\mathcal{C}$ because Γ needs to be recomputed. To reduce the frequency of expensive coreset reconstructions, we simply buffer o in $\mathcal{C}^{\text{out}}$, postponing coreset reconstruction until $|\mathcal{C}^{\text{in}}| = |\mathcal{C}^{\text{out}}|$. Immediately following a reconstruction, $\mathcal{C}^{\text{out}} = \varnothing$ and $\mathcal{C} = \mathcal{C}^{\text{in}}$.

When reconstructing the coreset, we attempt to reuse the objects in the current coreset whenever possible. Let $\mathcal{C}'$ denote the content of $\mathcal{C}$ before reconstruction. Let $\mathcal{O}_g$ be the set of new k nearest neighbors for a grid point $g \in \mathcal{G}$. For each object $o \in \mathcal{O}_g \setminus \mathcal{C}'$, if there exists an object $o' \in \mathcal{C}' \setminus \mathcal{O}_g$, such that the distance from g to o' is approximately the same as the distance from g to o, we substitute o with o'. This technique reduces the number of changes in the coreset membership, which in turn helps reduce the cost of maintaining the data structures built on the coreset.

Object deletion. Suppose an object $o \in \mathcal{O}$ is deleted. If $o \notin \mathcal{C}$, there is nothing to do. If $o \in \mathcal{C}^{\text{out}}$, we simply delete o from $\mathcal{C}^{\text{out}}$. Next, suppose $o \in \mathcal{C}^{\text{in}}$.

If o is not an anchor point in $\mathcal{A}$ defining the affine transform Γ, let $\mathcal{G}_o$ denote the subset of grid points $g \in \mathcal{G}$ such that $\Gamma(o)$ is one of g's approximate k nearest neighbors. We delete o from $\mathcal{C}^{\text{in}}$, and for each $g \in \mathcal{G}_o$, we compute the approximate k-th nearest neighbor of g and add it to $\mathcal{C}^{\text{in}}$.

If o happens to be an anchor point in $\mathcal{A}$, we need a new affine transform. Thus, we trigger the reconstruction of the coreset. Again, as discussed in the case of object insertion, we attempt to reuse the objects in the current coreset whenever possible.

4.3 Updating Indexes and Top-k Lists

Recall from Section 3.2 that we maintain a number of indexes for scalable processing of continuous top-k queries. For example, the preference-driven approach maintains an index $\mathcal{I}$ of objects (for preference top-k queries) and an index $\mathcal{J}$ of cutoff points. The hybrid approach maintains $\mathcal{I}$ and a search tree $\mathcal{T}$. With the coreset

approach, these indexes are now based on $\mathcal{C}$ instead of $\mathcal{O}$. When $\mathcal{C}$ changes, we need to update these indexes as well as the approximate top-k lists for all preferences. Naively, we can simply update them for every change to $\mathcal{C}$, but this strategy is expensive because a single insertion or deletion in $\mathcal{O}$ may sometimes translate to many changes to $\mathcal{C}$, as discussed in Section 4.2. The key observation is that we do not have to carry out some updates to the indexes and top-k lists immediately. To illustrate, suppose that the insertion of an object o from $\mathcal{O}$ causes another object o' to disappear from $\mathcal{C}$. There is no need to remove o' from the top-k list of a preference, because o' has not been deleted from $\mathcal{O}$, and the old list will continue to serve correctly as an approximate top-k list. Likewise, there is no need to delete o' from the indexes.

Therefore, following this intuition, we use a lazy approach to update the indexes and the top-k lists. We maintain two buffers:

- *Deletion buffer* stores the set ∇ of objects that have been deleted from $\mathcal{C}$ but not from $\mathcal{O}$; these objects are still present in the index $\mathcal{I}$ of objects and the tree structure $\mathcal{T}$.
- *QRS buffer* stores a set Δ of objects that have been inserted into $\mathcal{C}$ because of other object updates (i.e., o itself was already present in $\mathcal{O}$ before it is inserted into $\mathcal{C}$); these objects are inserted into $\mathcal{I}$ and $\mathcal{T}$, but they are not used to update the index $\mathcal{J}$ of cutoff points or the top-k lists.

We now describe the procedures for updating the indexes and top-k lists when an object is inserted or deleted in $\mathcal{O}$. The description below covers the maintenance of $\mathcal{I}$, $\mathcal{J}$, and $\mathcal{T}$; in practice, we only need to maintain the subset of these indexes used by the approach chosen from Section 3.2.

Object insertion. Suppose a new object o is inserted into $\mathcal{O}$. Recall the coreset update algorithm in Section 4.2. If o does not affect $\mathcal{C}$, there is nothing to do and we stop. If o is added to $\mathcal{C}$, we insert it into $\mathcal{I}$ and $\mathcal{T}$. We compute the set of affected preferences as discussed in Section 3, and update their top-k lists as well as their corresponding cutoff points in $\mathcal{J}$.

Furthermore, if the insertion of o causes a set $\mathcal{C}^-$ of objects to be removed from $\mathcal{C}$, we insert $\mathcal{C}^-$ into the deletion buffer ∇ and avoid updating $\mathcal{I}$ and $\mathcal{T}$.

Finally, if the insertion of $\mathcal{O}$ causes a set $\mathcal{C}^+$ of objects to be added to $\mathcal{C}$ (which happens when $\mathcal{C}$ is reconstructed), we process each $o' \in \mathcal{C}^+$ as follows. If $o' \in \nabla$, we simply remove it from ∇ and we are done; otherwise, we insert o' into $\mathcal{I}$, $\mathcal{T}$, and the QRS buffer Δ, without updating $\mathcal{J}$ or any top-k lists.

Object deletion. Suppose an existing object o is deleted from $\mathcal{O}$. If o is in neither the current coreset nor the deletion buffer ∇, we simply stop. Otherwise, we delete o from there and from $\mathcal{I}$ and $\mathcal{T}$. We also compute the set of affected preferences, and update their top-k lists as well as their corresponding cutoff points in $\mathcal{J}$.

Recall the coreset update algorithm in Section 4.2. If o was in the $\mathcal{C}$ and the coreset is not reconstructed, then the deletion of o can cause insertion of a set $\mathcal{C}^+$ of objects into $\mathcal{C}$. As in the case of object insertion discussed above, for each $o' \in \mathcal{C}^+$, if $o' \in \nabla$, we simply remove it from ∇; otherwise, we insert o' into $\mathcal{I}$, $\mathcal{T}$, and the QRS buffer Δ, again without updating $\mathcal{J}$ or any top-k lists.

Finally, if $\mathcal{C}$ was reconstructed as the result of deleting o, let $\mathcal{C}^+$ denote the set of objects inserted into $\mathcal{C}$ and let $\mathcal{C}^-$ denote the set of objects deleted from $\mathcal{C}$. We insert each object of $\mathcal{C}^-$ in the deletion buffer ∇. The processing of $\mathcal{C}^+$ is more involved. Let $\mathcal{O}' = (\mathcal{C}^+ \setminus \nabla) \cup \Delta$. We insert the objects in $\mathcal{C}^+ \setminus \nabla$ into $\mathcal{I}$ and $\mathcal{T}$, and delete those in $\mathcal{C}^+ \cap \nabla$ from ∇. By performing a reverse top-k query for each object in $\mathcal{O}'$, we identify the set $\mathcal{Q}_o$ of preferences that need updating. We update their top-k lists as well as their corresponding cutoff points in $\mathcal{J}$. Further details are omitted.

5 Experimental Evaluation

Approaches compared. For static reverse top-k queries, we have implemented our approach based on halfspace range queries (Section 3.1) using a quad-tree as the underlying index for cutoff points;[7] we call this algorithm ***HSR*** for short. For comparison, we implemented the *RTOP-Grid* algorithm by Vlachou et al. [28], which is the most recent and most relevant to our work; we call this algorithm ***GRID*** for short.

For the problem of processing a large number of continuous top-k queries, we have implemented all three approaches discussed in Section 3.2: preference-based, QRS-based, and hybrid. We do not compare with GRID in this case, because GRID does not handle object updates efficiently, and is already significantly outperformed by our approach in the static case (as we will see in Section 5.1).

For the three approaches, we again use quad-trees for the underlying indexes when applicable. For the QRS-based approach, we use a quad-tree to store the QRS, stopping when a node v's bounding box B_v is strictly above or below the QRS, or intersects fewer than τ_n hyperplanes in $\mathcal{O}^*$—analogous to the hybrid approach in Section 3.2.3 with $\tau_m = 0$ such that there are no grey-sparse nodes.

Finally, we have also implemented the coreset-based approach for the approximate version of the problem. All algorithms are implemented in C++.

Performance metrics. We consider the following when evaluating competing approaches:

- ***Time (per request):*** The wall-clock time for handling a request, be it a reverse top-k query in the static case, or an object or preference update in the dynamic case (which includes maintenance of data structures, processing of affected preferences, etc.).
- ***# calls:*** The number of calls to query primitives—halfspace range or top-k queries—discussed in Section 2.3. This metric allows performance to be measured independent from particular implementations of the primitives.
- ***Approximation error (estimated):*** The relative error observed in the answers produced by our coreset-based approximation approach in Section 4. For a coreset $\mathcal{C} \subseteq \mathcal{O}$, we measure the error in the top-k answer for preference q as

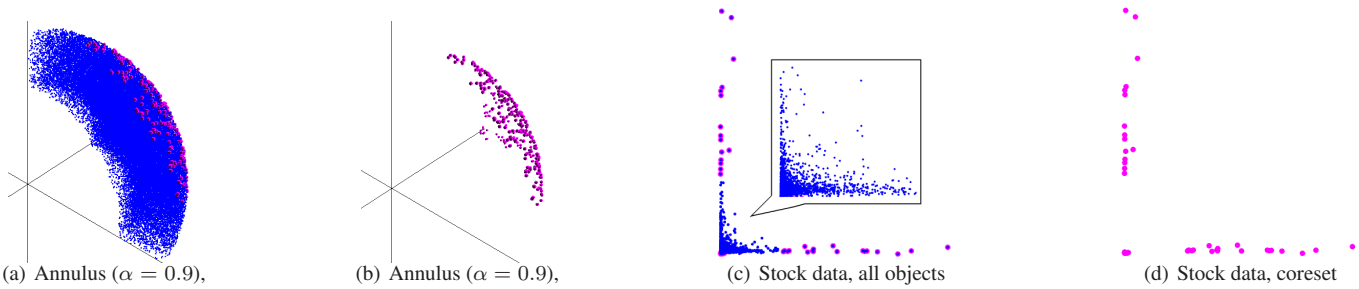
$$\max_{i \in \{1,\ldots,k\}} 1 - \frac{\langle q, \pi_{\leq i}(q, \mathcal{C})\rangle}{\langle q, \pi_{\leq i}(q, \mathcal{O})\rangle}.$$

  This measure is more stringent than what we bound in (1) for our definition of approximation; it in fact corresponds to the stronger definition of approximation in (2) in Section 4.1. To estimate the average error when $\mathcal{Q}$ is large or unknown, we randomly choose 1,000 preferences and compute the average.

Experiments were conducted on a Dell OptiPlex 990 with 3.40GHz Intel Core i7-2600 CPU, 8M cache, and 8GB memory.

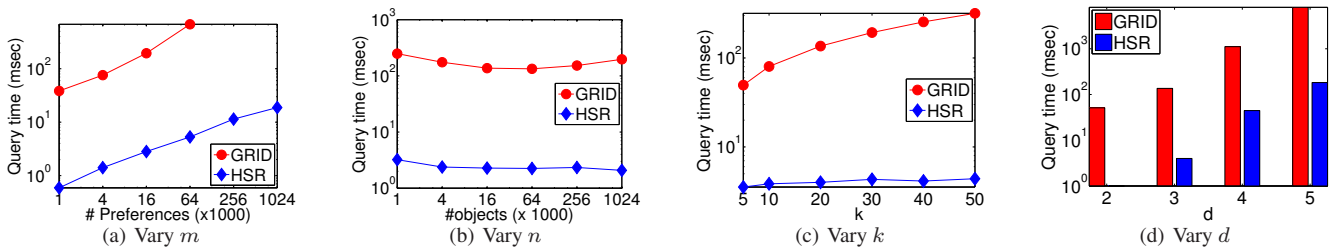
Workloads. We have experimented with a number of synthetic and real object workloads. We choose to focus on the results for the synthetic ***annulus-uniform*** and ***annulus-clustered***, because they enable testing with a wide range of data characteristics. Objects are drawn from the portion inside the positive orthant of an annulus in $\mathbb{R}^d$ centered at the origin with outer radius 1 and inner radius $\alpha \in [0, 1]$. For annulus-uniform, objects are uniformly distributed inside the annulus. For annulus-clustered, objects are distributed across a mixtures of 20 Gaussians (clipped to the annulus); parameters of the Gaussians allow further control of the clusteredness.

[7] We have also experimented with a kd-tree implementation, which showed comparable performance: it works better than the quad-tree for $d > 4$ and worse for $d < 4$. Since the choice does not change any conclusion drawn in this section, we do not show results for the kd-tree in this paper.



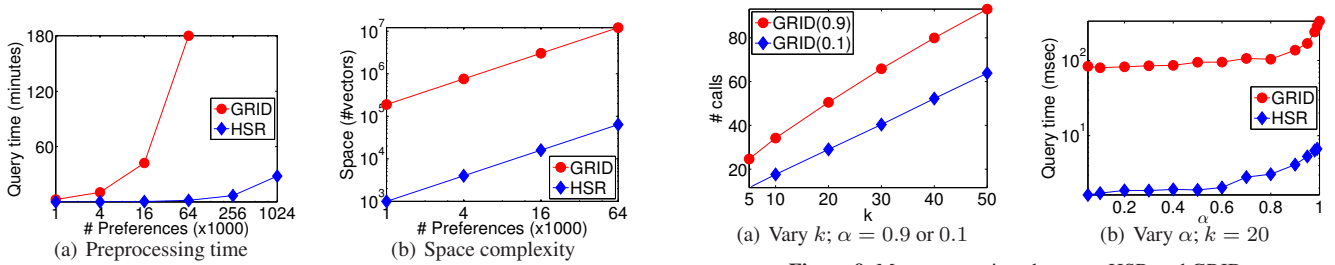
(a) Annulus ($\alpha = 0.9$), all objects (b) Annulus ($\alpha = 0.9$), coreset (c) Stock data, all objects (d) Stock data, coreset

Figure 6. Illustration of object workloads.



(a) Vary m (b) Vary n (c) Vary k (d) Vary d

Figure 7. Comparison between HSR and GRID (previous approach) for static reverse top-k queries; $\alpha = 0.9$.



(a) Preprocessing time (b) Space complexity (a) Vary k; $\alpha = 0.9$ or 0.1 (b) Vary α; $k = 20$

Figure 8. Additional scalability comparison between HSR and GRID.

Figure 9. More comparison between HSR and GRID.

For example, Figure 6(a) shows a set of objects $\mathcal{O}$ from annulus-uniform with $\alpha = 0.9$, and Figure 6(b) illustrates a coreset for $\mathcal{O}$.

To generate an object update for the workload, we first choose either insertion or deletion with equal probability. For insertion, we draw a new object from the same distribution used to draw the initial object set. For deletion, we choose an existing object at random with equal probability.

Annulus-uniform and annulus-clustered are related to the synthetic workloads (*correlated*, *anti-correlated*, and *uniform*) described in [11]. Note that α gives us some control over the "hardness" of the problem. As α approaches to 0, more and more objects, particularly those closer to the origin, do not participate in any top-k lists. The object distribution becomes uniform inside the ball. It remains harder for the problem than uniform distribution inside the unit box, for which only few objects close to the corners of the unit box participate in any top-k lists. The distribution of objects for annulus-clustered, generated with one single Gaussian, resembles correlated. As α approaches 1, the objects lie on the sphere, the distribution of objects becomes more anti-correlated, and any object can appear in a top-k list; this is in some sense captures the worst-case behavior.

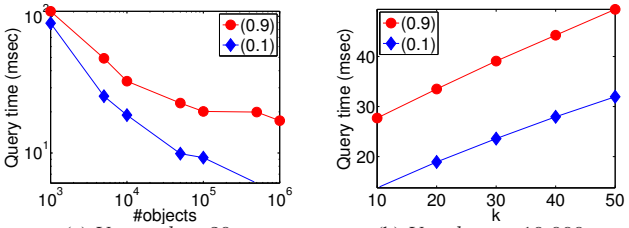
In addition to synthetic object workloads, we have obtained data for 2,374 stocks on NYSE and NASDAQ from Yahoo! Finance. For each stock, we collected its estimated earnings per stock (EPS) and weekly historical quotes (opening, closing, lowest, and highest prices and volume) in 2011. EPS can be used to convert each price to a price-to-earning ratio (PER), which is a more normalized metric than raw price for comparing different stocks. In our experiments we use only 2 dimensions: volume, and PER based on the closing price (although PER can be generated from various available prices, they would be extremely similar). Figure 6(c) shows the objects from this dataset, and Figure 6(d) shows its coreset. Note that the extreme points are well-represented in the coreset, while the cluster near the origin requires few representatives.

Preferences are generated from one of the following two distributions. With ***Uniform***, we draw preference uniformly at random from the unit sphere $\mathbb{S}^{d-1}$ inside the positive orthant of $\mathbb{R}^d$. With ***Clustered***, preferences are distributed across a mixtures of 20 Gaussians over the sphere; parameters of the Gaussians allow further control of the clusteredness.

5.1 Static Reverse Top-k Queries

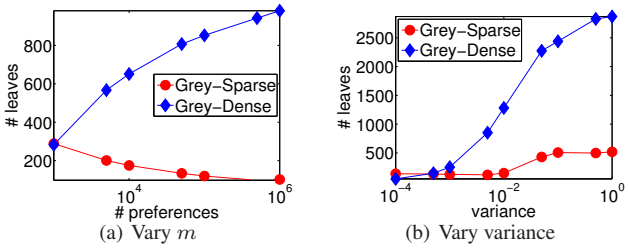
First, we compare our solution, HSR, with GRID [28]. Unless specified otherwise, $d = 3$, $m = 10{,}000$, $n = 10{,}000$, and $k = 20$ in this section. The objects are generated from annulus-uniform with $\alpha = 0.9$ (unless specified otherwise). One thousand query objects are drawn from the same distribution.

Figures 7(a) and 7(b) compare the average query time as we vary m (the number of preferences) and n (the number of objects), respectively. Our algorithm, HSR, which uses a linear-size data structure, performs one to two orders of magnitude better than GRID. We do not show results for GRID when $m = 256{,}000$ and 1,024,000, because it becomes too expensive. HSR's scalability advantage over GRID is reflected not only in terms of query time, but also preprocessing time (Figure 8(a)) and space consumption (Figure 8(b)). Preprocessing for GRID involves reordering of preferences and many reverse top-k computations for materialized



(a) Vary n; $k = 20$ (b) Vary k; $n = 10{,}000$

Figure 10. Reverse top-k query on 1 million preferences; $d = 3$.



(a) Vary m (b) Vary variance

Figure 11. Numbers of grey-sparse and grey-dense leaves in $\mathcal{T}$.

views, and it takes more than 3 hours for $m > 64{,}000$. GRID also consistently uses two orders of magnitude more space than HSR; here, we measure the space of GRID by the number of preferences it materializes, and the space of HSR by the number of cutoff points it indexes, both of which are d-dimensional vectors.

Figure 7(c) shows that when we vary k, the average query time increases for GRID but remains almost the same for HSR. The reason is that the number of top-k queries made by GRID depends on k, which becomes clear when we examine Figure 9(a). Figure 9(a) shows the number of top-k queries made by GRID for $\alpha = 0.9$ (the workload used by Figure 7(c)), and the number for $\alpha = 0.1$. Both exhibit growth linear in k, and we see that $\alpha = 0.9$ is indeed "harder" than $\alpha = 0.1$. On the other hand, HSR always issues one halfspace range query per request (and therefore it is not shown in Figure 9(a)), regardless of k and α.

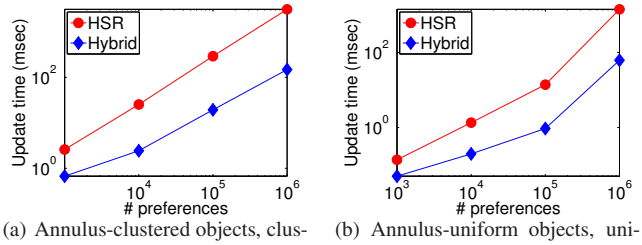
Figure 7(d) shows how HSR and GRID perform when we increase dimensionality. We see that the advantage of HSR over GRID is maintained as we increase d. For $d = 2$, HSR answers a reverse top-k query in 0.38 milliseconds on average, which, if shown in Figure 7(d), would have been below the horizontal axis (note the log-scaled vertical axis).

Figure 9(b) summarizes the comparison between HSR and GRID we when vary α, the inner radius of the annulus. The query time generally increases as the annulus becomes thinner (approaching a sphere), but HSR maintains its lead over GRID across all α values.

Figures 10 shows the performance of HSR when the number of preferences scales up to one million. GRID becomes too slow to run in this case. Two curves are shown in Figure 10(a): one for $\alpha = 0.1$ and one for $\alpha = 0.9$. For $\alpha = 0.1$, the algorithm performs better when n is bigger, because more objects actually lower the chance that a query object becomes relevant to the preferences. Figure 10(b) shows the average query time slightly increases as k increases. Compared with Figure 7(c), the average query time increases by a factor of 10 when the number of preferences increases by a factor of 100. The reason is that the size of the index for halfspace range queries depends on the number of preferences.

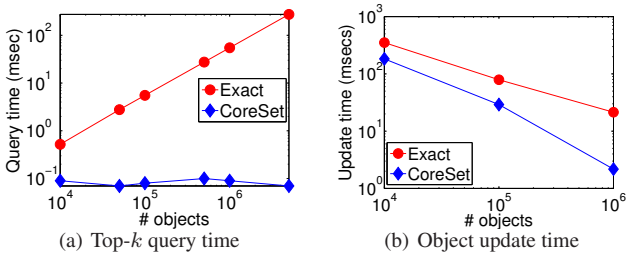
5.2 Continuous Top-k Queries

An object update is costly for GRID because a lot of materialized views need to be recomputed for the object update. Many top-k queries are called as subroutines even if the object update does



(a) Annulus-clustered objects, clustered preferences (b) Annulus-uniform objects, uniform preferences

Figure 12. Preference-driven vs. hybrid.



(a) Top-k query time (b) Object update time

Figure 13. Exact vs. coreset-based approximation.

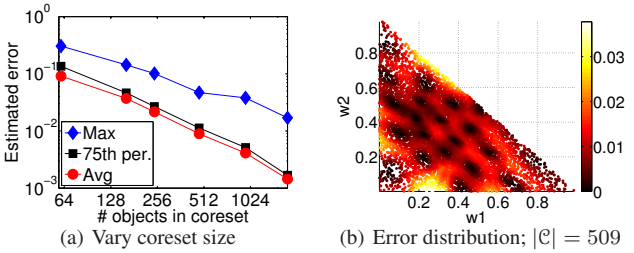
not affect any preference's top-k results. For annulus-uniform with $\alpha = 0.8$, $d = 2$, $m = 10{,}000$, $n = 1{,}000$, and $k = 10$, the average update time of GRID is 40 seconds. In comparison, HSR takes only 0.014 seconds per update. Since the performance of HSR clearly dominates that of GRID for continuous top-k queries, we omit GRID in the remainder of this section. Unless specified, $\alpha = 0.8$, $k = 10$, $d = 2$, objects are drawn from annulus-uniform, and preferences generated from the clustered distribution. For hybrid approach, we set $\tau_m = 1$ and $\tau_n = 1$ (recall Section 3.2.3).

We first see how the hybrid approach automatically adapts to the object and preference workloads. Figure 11(a) shows that as the number of preferences increases, more grey-sparse nodes in $\mathcal{T}$ are converted into grey-dense nodes. Those preferences in grey-dense nodes are not processed individually, making hybrid approach more scalable to a large number of preferences. Figure 11(b) shows the number of grey-sparse and grey-dense leaves as we vary the variance of the Gaussian distributions from which preferences are generated. When variance is small, many parts of the QRS have few or no preferences, so hybrid uses fewer grey-dense nodes and takes a preference-driven approach for these parts.

Next, we compare the preference-driven and hybrid approaches for continuous top-k queries. We set the number of objects to 1,000, and vary the number of preferences from 1,000 to one million. Figures 12(a) and 12(b) compare the performance of the preference-driven (denoted HSR in figures) and hybrid approaches for annulus-clustered and annulus-uniform, respectively; preferences are drawn from clustered and uniform, respectively. We omit the combinations of (annulus-clustered objects, uniform preferences) and (annulus-uniform objects, clustered preferences) because they show similar trends. For these workloads, when the ratio between the number of preferences and the number of objects becomes large, the hybrid approach performs significantly better than the preference-driven one, as it avoids multiple computations for many preferences sharing the same k-th ranked object.

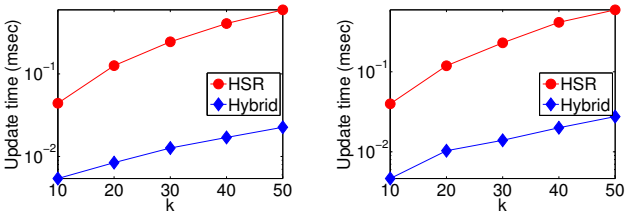
5.3 Continuous Approximate Top-k Queries

In this section, the number of preferences is set to 100,000; $k = 20$, $d = 3$, and $\alpha = 0.9$. Figure 13 shows the effect of the number of input objects on the coreset-based approximation algorithm, in comparison with the exact one; here, the size of the coreset is fixed



(a) Vary coreset size (b) Error distribution; $|\mathcal{C}| = 509$

Figure 14. Approximation error; $|\mathcal{O}| = 100{,}000$.



(a) Clustered preferences (b) Uniform preferences

Figure 15. Preference-driven vs. hybrid: Yahoo! Finance data; $m = 100{,}000$.

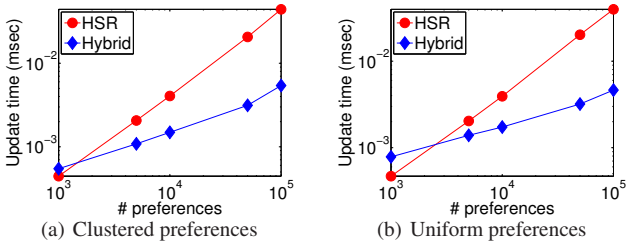
roughly at 1,000. Figure 13(a) shows the top-k query time when the number of objects varies from 10,000 to one million. While the query time for the exact algorithm increases, it remains roughly the same for the coreset-based algorithm, because the size of the top-k index is proportional to $|\mathcal{C}|$ instead of $|\mathcal{O}|$. Since an insertion or deletion of a preference involves a top-k query, the coreset-based algorithm will be able to handle preference updates better than the exact one for a large set of objects. Figure 13(b) shows the processing time per object update when the number of objects varies from 10,000 to one million. The gap between the performances of the exact and coreset-based approximation algorithms widens as the number of objects increases, because 1) when $|\mathcal{O}|$ becomes large, most object updates would not affect the coreset if an object update is randomly chosen, and 2) the top-k query can be answered more efficiently on a smaller coreset.

Figure 14(a) shows how the size of the coreset affects the quality of the approximation. As expected, the larger the coreset, the higher the accuracy. By choosing roughly 500 objects in the coreset, the estimated maximum and average errors are less than 0.05 and 0.01, respectively. Moreover, majority of the errors are small, as indicated by the closeness between the average error and error at the 75th percentile. Figure 14(b) further plots the distribution of errors over the preferences in $\mathcal{Q}$. Preferences at the boundary tend to have slightly higher approximation errors.

5.4 Yahoo! Finance Data

Experiments in this sections study the performance of our exact algorithms (preference-based and hybrid) on the object (stock) data collected from Yahoo! Finance. Since user data are not disclosed to the public, we use synthetic preferences generated from clustered and uniform. While the distribution of objects and update workload are considerably different from the synthetic ones (as illustrated in part by Figure 6), we see similar performance results as Section 5.2.

Figure 15 shows the average update time as we vary k. As expected, the number of affected preferences increases as k increases. For the preference-based approach, a top-k query is called for each affected preference. For the hybrid approach, the complexity of the k-level depends on the value of k, and a larger k increases the size of the search tree $\mathcal{T}$. Figure 16 shows the average update time as we increase the number of preferences. Similar to the synthetic workloads (Figure 12), all curves exhibit growth proportional to m, because the number of affected preferences increases as m increases.



(a) Clustered preferences (b) Uniform preferences

Figure 16. Preference-driven vs. hybrid: Yahoo! Finance data; $k = 10$.

For a small number (up to around a thousand) of preferences, the preference-based approach may be more attractive because of the overhead of hybrid's flexibility and the small workload size in this case, but hybrid remains the better choice if m is reasonably large.

6 Related Work

There is a large body of literature on top-k query processing (see [20] for a survey); much of it concerns the linear preference top-k queries and variants [15, 19, 27, 24, 18, 17, 28] that we consider. Here we elaborate on three pieces of work that are most related to ours.

Das et al. [17] considered the problem of supporting ad hoc (i.e., non-continuous) top-k queries over streams. They also took a geometric approach and developed a data structure based on maintaining an arrangement of lines in the dual space. Their solution uses halfspace range queries as a primitive, but not for the purpose of solving reverse top-k queries as we propose. Time and space complexities are improved by pruning the set of objects to a superset of the k-skyband, which is computed by partitioning the arrangement into "strips" and using the top-k query results for the borders of the strips to prune dual lines from each strip. The query and update operations take linear time, and heuristics are required in choosing the partitioning. There was some discussion on the case of $d > 2$, but the solution was only evaluated for $d = 2$. In comparison, our coreset-based approach to approximating the k-level provides guarantees and generalizes to higher dimensions.

Mouratidis et al. [24] proposed the *TMA* algorithm (and the more specialized *SMA*) for supporting multiple continuous top-k queries over data streams. TMA partitions the primal space into grids, and for each cell, stores an "influence list" of queries (those returned by a reverse top-k query with the cell's top-right corner). Given an object update, TMA identifies affected queries by searching for affected cells in an order that minimizes the number of cells visited. Since TMA materializes the top-k answer for each query, it requires more space than our approach of recording only the cutoff points. They also target fewer number (thousands) of queries than our work (hundreds of thousands). A direct comparison with our work is difficult because TMA and SMA also have features specific to the object update pattern under the sliding-window semantics.

Most relevant to this paper is the work by Vlachou et al. [28]. Their *monochromatic* reverse top-k algorithm can compute, without knowing the actual preferences, a description of the set of possible preferences that would be affected by a given object. The algorithm works for $d = 2$, based on similar observations as *ranked join indices* [27]. Our QRS-based framework and techniques can solve the same problem in higher dimensions as well; see Section 7 for more details. For the *bichromatic* reverse top-k problem, where the set of preferences is given, two algorithms were proposed. *RTA* heuristically orders the preferences to be processed based on similarity, to increase the chance that the top-k query result for the current preference can be reused for the next preference. *RTOP-Grid* uses a grid data structure for pruning. For each cell, a reverse top-k query is run for the lower-left and upper-right corners, and the re-

sult lists are stored in the cell. These lists are used to reduce the set of preferences to be further evaluated using RTA. RTOP-Grid provides no theoretical performance guarantees, and object updates are particular expensive for the grid data structure. Our experimental evaluation in Section 5 compares RTOP-Grid with our solutions.

7 Conclusion and Other Applications

In the paper, we studied the problem of scalably processing a large number of continuous top-k queries, each with a different preference vector for ranking multi-attribute objects. We proposed the notion of QRS (query response surface) and developed our solutions within a geometric framework. Recognizing the connection to halfspace range queries, we obtained data structures for reverse top-k queries with linear space and sublinear query time. Building on this result, we developed a fully dynamic solution supporting both object and preference updates efficiently. We also defined and solved an approximate version of the problem, further improving efficiency with little loss of accuracy. Experimental evaluation confirmed the effectiveness of our ideas such as selective QRS-driven processing and coreset-based QRS simplification, which helped advance our solutions in both scalability and functionality.

In closing, we briefly discuss several settings beyond those focused on by this paper, where our techniques may be applicable.

Reverse k-nearest-neighbor queries have been widely studied by the database community; see [25] for an overview. Though these queries are not our focus here, we briefly point out how they can be handled by our approach. More precisely, a set $\mathcal{O}$ of points in $\mathbb{R}^d$ can be mapped to a set $\hat{\mathcal{O}}$ of points in $\mathbb{R}^{d+1}$ so that the k-nearest-neighbor query for a point $q \in \mathbb{R}^d$ can be formulated as the top-k query for a preference $\hat{q} \in \mathbb{S}^d$; we omit the details from here.

In some settings, we do not have the set $\mathcal{Q}$ of preferences explicitly. Instead, given an object update, we are interested in obtaining (a description of) the set of all possible preferences affected by it. This query is termed *monochromatic reverse top-k* by [28], with applications in business analysis [28] and in publish/subscribe systems using the *message reformulation paradigm* [14]. Our concept of QRS and the QRS-driven approach in Section 3.2 offer a solution that generalizes to high dimensions. Maintaining the full QRS is expensive, however. When approximation is acceptable, our coreset-based approach in Section 4 can simplify the QRS, improve running time, and reduce the complexity in describing the affected preferences.

Recently, there is growing interest in handling uncertainty in preference vectors and assessing sensitivity in ranking to perturbations in preferences [26]. Our notion of QRS provides a natural framework for these problems, and our coreset-based approximation can be readily applied to improve solution scalability. Further investigation would be a promising direction of future work.

Finally, preference top-k queries also have applications in information retrieval (where, e.g., a multi-keyword search can be seen as a preference top-k query over documents in a high-dimensional keyword vector space) and in information integration (where results from multiple sources are merged and ranked according to a preference function). Recent work [21] has studied how to share the work involved in processing multiple such queries. It would be interesting to investigate whether our techniques can be applied help improve scalability in a complementary manner.

8 References

[1] P. Afshani and T. M. Chan. Optimal halfspace range reporting in three dimensions. In *SODA '09*, pages 180–186, 2009.

[2] P. K. Agarwal, L. Arge, J. Erickson, P. G. Franciosa, and J. S. Vitter. Efficient searching with linear constraints. *J. Comput. Syst. Sci.*, 61(2):194–216, 2000.

[3] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Approximating extent measures of points. *Journal of the ACM*, 51:606–635, 2004.

[4] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Geometric approximation via coresets. In J. E. Goodman, J. Pach, and E. Welzl, editors, *Combinatorial and Computational Geometry*, pages 1–30. Cambridge University Press, New York, 2005.

[5] P. K. Agarwal, S. Har-Peled, and H. Yu. Robust shape fitting via peeling and grating coresets. In *SODA '06*, pages 182–191, 2006.

[6] P. K. Agarwal and J. Matoušek. Dynamic half-space range reporting and its applications. *Algorithmica*, pages 325–345, 1995.

[7] P. K. Agarwal, J. M. Phillips, and H. Yu. Stability of *epsilon*-kernels. In *ESA'10*, pages 487–499, 2010. Springer-Verlag.

[8] P. K. Agarwal and M. Sharir. Arrangements and their applications. In *Handbook of Computational Geometry*, pages 49–119. Elsevier Science Publishers B.V. North-Holland, 1998.

[9] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[10] G. Barequet and S. Har-Peled. Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. In *SODA '99*, pages 82–91, Philadelphia, PA, USA, 1999.

[11] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE'01*, pages 421–430, Washington, DC, USA, 2001.

[12] H. Brönnimann, B. Chazelle, and J. Pach. How hard is half-space range searching. *Discrete & Comput. Geom.*, 10:143–155, 1993.

[13] T. M. Chan. Three problems about dynamic convex hulls. In *SoCG'11*, pages 27–36, New York, NY, USA, 2011.

[14] B. Chandramouli, J. Xie, and J. Yang. On the database/network interface in large-scale publish/subscribe systems. In *SIGMOD'06*, pages 587–598, Chicago, Illinois, USA, June 2006.

[15] 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. *SIGMOD Rec.*, 29:391–402, May 2000.

[16] B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, June 1985.

[17] G. Das, D. Gunopulos, N. Koudas, and N. Sarkas. Ad-hoc top-k query answering for data streams. In *VLDB'07*, pages 183–194, 2007.

[18] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *VLDB '06*, pages 451–462. 2006.

[19] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: a system for the efficient execution of multi-parametric ranked queries. *SIGMOD Rec.*, 30:259–270, May 2001.

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

[21] M. Jacob and Z. G. Ives. Sharing work in keyword search over databases. In *SIGMOD '11* , pages 577–588, 2011.

[22] J. Matousek. Reporting points in halfspaces. *Comput. Geom.*, 2:169–186, 1992.

[23] J. Matousek. *Lectures on Discrete Geometry*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[24] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD '06*, pages 635–646, New York, NY, USA, 2006.

[25] D. Papadias and Y. Tao. Reverse nearest neighbor query. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 2434–2438. Springer, 2009.

[26] M. A. Soliman, I. F. Ilyas, D. Martinenghi, and M. Tagliasacchi. Ranking with uncertain scoring functions: semantics and sensitivity measures. In *SIGMOD '11*, pages 805–816, 2011.

[27] P. Tsaparas, N. Koudas, Y. Kotidis, T. Palpanas, and D. Srivastava. Ranked join indices. In *ICDE '03*, pages 277–288, 2003.

[28] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *ICDE '10*, pages 365–376, 2010.

[29] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Monochromatic and bichromatic reverse top-k queries. *IEEE Trans. Knowl. Data Eng.*, 23(8):1215–1229, 2011.

[30] H. Yu, P. K. Agarwal, R. Poreddy, and K. R. Varadarajan. Practical methods for shape fitting and kinetic data structures using core sets. In *SCG '04*, pages 263–272, 2004.