Top-k Preferences in High Dimensions

Albert Yu[1], Pankaj K. Agarwal[2], Jun Yang[3]

Department of Computer Science, Duke University

{[1]syu,[2]pankaj,[3]junyang}@cs.duke.edu

***Abstract*—Given a set of objects $\mathcal{O}$, each with d numeric attributes, a *top-k preference* scores these objects using a linear combination of their attribute values, where the weight on each attribute reflects the interest in this attribute. Given a query preference q, a *top-k query* finds the k objects in $\mathcal{O}$ with highest scores with respect to q. Given a query object o and a set of preferences $\mathcal{Q}$, a *reverse top-k query* finds all preferences $q \in \mathcal{Q}$ for which o becomes one of the top k objects with respect to q. Previous solutions to these problems are effective only in low dimensions. In this paper, we develop a solution for much higher dimensions (up to high tens), if many preferences exhibit *sparsity*—i.e., each specifies non-zero weights for only a handful (say 5–7) of attributes (though the subsets of such attributes and their weights can vary greatly). Our idea is to select carefully a set of low-dimensional *core subspaces* to "cover" the sparse preferences in a workload. These subspaces allow us to index them more effectively than the full-dimensional space. Being multi-dimensional, each subspace covers many possible preferences; furthermore, multiple subspaces can jointly cover a preference, thereby expanding the coverage beyond each subspace's dimensionality. Experimental evaluation validates our solution's effectiveness and advantages over previous solutions.**

I. Introduction

When examining a large number of objects with multiple attributes, users often find it useful to rank the objects according to personal preferences in order to focus on the top-ranking ones. A natural way of specifying this ranking is with an object scoring function whose parameters are set according to a user's preference. A simple but effective scoring function is a linear combination of the attribute values, where the weight associated with each attribute reflects the users' interest in this attribute. For example, consider an NBA player database that maintains, for thousands of players, various performance statistics such as points, rebounds, and assists per game. A user interested in tracking offensive players may care most about points and assists, and hence place larger weights on these attributes. On the other hand, a user interested in tracking defensive players may care about steals and blocks, and hence place a small or zero weight on assists. There has been much work on *preference top-k queries* [1], [2], [3], [4], [5], [6], [7], [8], [9], which return the top k objects ranked according to linear scoring functions.

Also of interest are *reverse preference top-k queries* [10], [8], [11]. Here, we have a set of objects and a set of top-k preferences. Given a new object, we want to know for which preferences this new object will enter their result top k objects. These queries have applications in publish/subscribe [8] (e.g., monitoring top players over time) and market research [10] (e.g., what-if analysis of public interest in a new product).

Challenge: curse of dimensionality. Supporting linear preference top-k queries and the reverse top-k queries becomes challenging even for moderate dimensions (say 20). The best known data structures, with provable performance bounds, answer a query in $O(\log n)$ time using roughly $n^{\lfloor d/2 \rfloor}$ space or in roughly $n^{1-1/\lfloor d/2 \rfloor}$ time using linear space [12]. In terms of practical data structures, the Threshold Algorithm (TA) [13] is efficient if every top-k object is ranked high in at least one dimension. However, as the dimensionality d grows, there is a higher chance that an object has a low rank even if it ranks high along one dimension. The layer-based approach, represented by [1], indexes layers of convex hulls for the objects in the full-dimensional space; computing a convex hull takes $O(n^{\lfloor d/2 \rfloor} + n\log(n))$ time, and the outer layers grow in size quickly with d, even if points are uniformly distributed in a ball. The view-based approach [14], [5] uses a set of materialized top-k views to compute top-k queries, but in high dimensions, a large number of materialized views are required to provide adequate support for queries. Recently, Heo et al. [7] combined the layer-based technique with TA-style dimension-wise filtering for top-k queries involving arbitrary subset of attributes. All work mentioned above tested their algorithms on data in at most 7 dimensions.

For reverse top-k queries, the approach of [10] reduces a reverse top-k query to m top-k queries, where m is the number of preferences in the worst case. Recently, a branch-and-bound algorithm is presented [11], but its R-tree index structure is not scalable in d. Our previous work [8] reduces a reverse top-k query to halfspace reporting. For $d \leq 3$, the query time is $O(\log n + t)$, using linear space, which is optimal; here t is the output size. For $d > 3$, if the storage requirement is near-linear, say $O(n\text{polylog}(n))$, then the query time of best known algorithms is $\Omega(n^{1-1/\lfloor d/2 \rfloor} + t)$ [12], where t is the number of results, and the hidden constant of proportionality is exponential in d; furthermore, these algorithms are too complex to implement. For practical data structures such as quad-trees and kd-trees, a halfspace query requires $\Omega(n)$ time in the worst case and roughly $O(n^{1-1/d} + t)$ for uniformly distributed points. Hence, existing approaches will not outperform a simple linear scan.

Opportunity: sparse preferences. We observe that in practice, even if data have high dimensionality, users are usually interested in only a small subset of attributes—we would not expect many users to specify preferences with a large number of attributes having non-zero weights. Thus, there is an opportunity to develop techniques for handling such "sparse" preferences differently from and more efficiently than the general case. If many preferences are sparse, we can improve overall performance by speeding up the common case.

One possible approach exploiting this observation is to use the existing dimensionality reduction techniques, such as principal component analysis (PCA), random projection, and low-distortion embedding techniques [15], which are usually applied to the object set. We argue that reducing object dimensionality alone is neither perfect nor complete. While these methods are effective in projecting data to moderate dimensions, say 100's to 10's, using them to project objects onto 5–7 dimensions creates significant error. Also, attributes in the reduced space are harder for users to work with as they may no longer have intuitive meanings. We can map preferences in the original space to ones in the reduced space, but they may become more difficult to handle as they may no longer retain their sparsity.

Our approach and results. We present efficient algorithms for top-k and reverse top-k queries in high dimensions. Our approach is effective when most preferences are sparse—i.e., each of them specifies non-zero weights for only a small number (say 2–6) of attributes (but they need not specify the same subset of attributes or similar weights on attributes). For top-k queries, in order to take advantage of sparsity in query preferences, our approach needs to assume the distribution of *which* attributes are specified by the preferences, but it still works well without accurate knowledge of the distribution of *what weights* are specified for these attributes.

Roughly, we follow a dimension-reduction framework, but we do not project all objects and preferences on a single low-dimensional subspace. Instead, we project them on many subspaces and, for each subspace, we index a subset of them.

In more detail, we carefully choose a set $\mathbb{H}$ of low-dimensional subspaces, called *core subspaces*, based on the given distribution of preferences. For each core subspace $H \in \mathbb{H}$, we choose a small subset of objects that are "relevant" for H and project them on H. Let $\mathcal{O}_H$ denote the resulting projections. Building on our techniques for handling low-dimensional preferences in [8], we index $\mathcal{O}_H$ for each H.

To answer a top-k query with respect to a sparse preference q, we choose a small subset $\Gamma_q \subset \mathbb{H}$ of core subspaces, which "cover" the query preference q. For each $H \in \Gamma_q$, we compute the top-βk ranked objects of $\mathcal{O}_H$ for a parameter $\beta \geq 1$, with respect to the preference q (or rather, w.r.t. the projection of q on H). Finally, we return the top k among these objects.

To support reverse top-k queries for a set $\mathcal{O}$ of objects and a set $\mathcal{Q}$ of preferences, we assign each preference $q \in \mathcal{Q}$ to a small subset of $\Gamma_q \subset \mathbb{H}$ of core subspaces that cover q. For each core subspace $H \in \mathbb{H}$, let $\mathcal{Q}_H$ denote the projections on H of preferences assigned to H. We index $\mathcal{Q}_H$ to support reverse top-βk queries against $\mathcal{O}_H$ and $\mathcal{Q}_H$. To answer a reverse top-k query for a query object o, we identify the core subspaces that are "relevant" for o, perform a reverse top-βk query with o in each of them, collect all result preferences, and filter out any false positives.

Our experimental evaluation confirms the effectiveness of our approach, which allows a desktop machine to handle hundreds of thousands of objects or preferences in 20 to 200 dimensions with speed and accuracy. To the best of our knowledge, our approach is the first to demonstrate this degree of scalability in both problem size and dimensionality.

Technical challenges and contributions. There are several technical challenges that we need to address to make our approach viable. First, how do we choose the core subspaces? A naive approach will be to make any subspace that contains some preferences to be a core subspace. For example, if we know that preferences specify non-zero weights for attribute subsets $\{1, 2\}$, $\{1, 3\}$, and $\{2, 3, 4\}$, then we make them core subspaces and build indexes for them: 2-dim indexes for $\{1, 2\}$ and $\{1, 3\}$, and 3-dim for $\{2, 3, 4\}$. This approach is not practical, however, because there are too many possible low-dimensional subspaces. For example, if objects have 20 attributes and each preference specifies at most three of them, one might have to build $\binom{20}{3} = 1{,}140$ different indexes.

Another possibility is to cluster the preferences into a small number of clusters and choose a representative preference, called a *view*, from each cluster. This view-based approach [14], [5] works if preferences are tightly clustered, objects are "well distributed," and the weights of query preferences for top-k queries follow the same distribution of $\mathcal{Q}$. As we will see later, this approach does not always work well, because each view is very "specific" and many more views will be needed as dimensionality grows. We show how to overcome the limitations of this approach with higher dimensional core subspaces, each of which effectively serves as a "super"-view that subsumes an infinite number of preference-based views lying in it. Section III describes our approach.

Second, it will be too expensive to build an index on the entire set of objects for each core subspace, so we describe a method (Section IV-A) for choosing a small set of objects to index. Analogously, it is expensive to index all preferences in each core subspace, so we introduce a method (Section IV-B) for assigning each preference to a small number of core subspaces where it will be indexed. Then, using the indexes we describe in Section IV, we show how to answer top-k and reverse top-k queries (Section V).

Finally, we cannot assume that all preferences are sparse or all can be covered by the selected core subspaces. Therefore, we also show in Section IV-C how to build full-dimensional indexes for uncovered preferences. In particular, we describe an approximation method similar to the one in [16], but with an improvement: if input objects lie on a low-dimensional surface, say of dimension τ, then we can choose a subset $\mathcal{C}$ of objects whose size is exponential only on τ, but polynomial in d, which provides top-k query answers that approximate those obtained by querying the entire set of objects.

II. Problem Statement

An *object* has d real-valued attributes and is represented as a point $(v_1, \ldots, v_d) \in \mathbb{R}^d$; the x_i-axis represents the i-th attribute. Let e_i denote the unit vector in direction x_i, i.e., the i-th coordinate of e_i is 1 and the rest are 0. A subset $I \subseteq [1, d]$ of attributes defines an axis-parallel subspace $\mathsf{Sp}(I)$ of $\mathbb{R}^d$ in which only the attributes of I have non-zero values. Formally, $\mathsf{Sp}(I) = \{\sum_{j \in I} \lambda_j e_j \mid \lambda_j \in \mathbb{R}\}$. For two axis-parallel subspaces $H_1 = \mathsf{Sp}(I_1)$ and $H_2 = \mathsf{Sp}(I_2)$, $\mathrm{span}(H_1, H_2)$ denotes the smallest axis-parallel subspace that contains both H_1 and H_2; equivalently, $\mathrm{span}(H_1, H_2) = \mathsf{Sp}(I_1 \cup I_2) = \{\lambda_1 x_1 + \lambda_2 x_2 \mid x_1 \in H_1, x_2 \in H_2, \lambda_1, \lambda_2 \in \mathbb{R}\}$.

A *preference* is represented as a unit vector in $\mathbb{R}^d$, i.e., a point $(w_1, \dots, w_d)$ on $\mathbb{S}^{d-1}$, the $(d-1)$-dimensional unit sphere embedded in $\mathbb{R}^d$ centered at the origin. Each $w_i \in [-1, 1]$ is the *weight* for the i-th attribute (weights can be negative). For a preference q, we define $\mathsf{Sp}(q)$ to be the subspace spanned by the non-zero attributes of q. Note that $\dim(\mathsf{Sp}(q))$ may be much smaller than d. For example, if $q = (1/\sqrt{2}, 1/\sqrt{2}, 0, \dots, 0)$, then $\mathsf{Sp}(q)$ is the 2-d x_1x_2-plane.

The *score* of an object o with respect to a preference q is $\langle q, o \rangle = \sum_{1 \le i \le d} w_i v_i$. 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. For a point $x \in \mathbb{R}^d$ and an axis-parallel subspace H, let x_H denote the projection of x on H. For example, if $x = (x_1, \dots, x_d)$ and H is spanned by attributes $\{1, 2, 4\}$, then $x_H = (x_1, x_2, x_4)$. For a preference q with $H = \mathsf{Sp}(q)$ and an object o, $\langle q, o \rangle = \langle q_H, o_H \rangle$; in other words, when computing the score of o w.r.t. q, it suffices to do so for their projections on the subspace $\mathsf{Sp}(q)$.

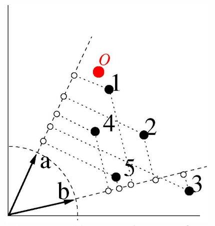


Fig. 1. Illustration of top-k queries in $\mathbb{R}^2$ (adapted from [8]). $\pi_{\le 5}(a, \mathcal{O}) = \langle 1, 2, 4, 3, 5 \rangle$; $\pi_{\le 5}(b, \mathcal{O}) = \langle 3, 2, 1, 5, 4 \rangle$.

Let $\mathcal{O} = \{o_1, o_2, \dots, o_n\} \subset \mathbb{R}^d$ denote the set of n objects of interest. For simplicity of exposition, we assume that no two objects have the same score for any preference we consider. Our framework and algorithms extend to handle ties in a straightforward manner. 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 $\pi_{\le i}(q, \mathcal{O}) = \{\pi_j(q, \mathcal{O}) \mid 1 \le j \le i\}$ 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 in the direction of q; see Figure 1. We are interested in:

(Preference) top-*k* query Given a query preference q, return $\pi_{\le k}(q, \mathcal{O})$.

Reverse (preference) top-*k* query Given a set of m preferences $\mathcal{Q} = \{q_1, q_2, \dots, q_m\}$ and a query object o, find the subset $\mathcal{Q}_o = \{q \in \mathcal{Q} \mid o \in \pi_{\le k}(q, \mathcal{O} \cup \{o\})\}$, i.e., all preferences in $\mathcal{Q}$ for which o is one of the top-k objects. We say that such preferences are *affected* by o.

III. Identifying Core Subspaces

In this section, we describe how to find the set $\mathbb{H}$ of core subspaces, which we use to build low-dimensional indexes. For these indexes to be practically efficient, we cap the maximum dimensionality of a core subspace at $\hat{\tau} = 5$.

Let $\mathcal{Q}$ be a set of preferences. It can be a set of given preferences for reverse top-k queries, or a past workload of forward top-k queries that informs index construction.

Our algorithm works in three stages. The first stage identifies the initial set $\mathbb{K}$ of *candidate* subspaces from the "sparse" preferences of $\mathcal{Q}$ (the formal definition of "sparseness" will follow shortly). If $\mathbb{K}$ is small, let $\mathbb{H} = \mathbb{K}$ and we are done. Otherwise, we proceed to the next stage, adding to $\mathbb{K}$ a few additional subspaces that span multiple subspaces of $\mathbb{K}$ and are "popular" (roughly speaking, a popular subspace can help "cover" many sparse preferences—the notion of "coverage" is intuitive but will be made more clear in Section IV-B). The last stage chooses a subset of $\mathbb{K}$ to cover most of the sparse preferences of $\mathcal{Q}$. We now describe each stage in detail.

Weight of a subspace. To capture the notion of "popularity," we define the *weight* of a subspace H (with respect to the set of sparse preferences $\mathcal{Q}_s$) as

$$w(H) = \sum_{q \in \mathcal{Q}_s} \|q_H\|^2 / (\dim(H))^\mu, \tag{1}$$

where q_H denotes the projection of q on H, and μ is a parameter (further explained below). Intuitively, the weight function favors those subspaces that have low dimensionality but preserve most information about preferences, in the sense that $\|q_H\|$ is large.

We choose $\|q_H\|^2$ instead of $\|q_H\|$ in this definition, because we wish to reward subspaces that preserve most information about a preference (i.e., $\|q_H\|$ is close to 1), and penalize those that preserve little information about a preference (i.e., $\|q_H\|$ is close to 0). For example, given two preferences, consider i) two subspaces, where each contains one preference (whose projection has norm of 1) but is orthogonal to the other preference (whose projection has norm 0), versus ii) two subspaces for which both preferences have projections of norm 0.5. Intuitively, the two subspaces in (i) are better because they provide "full coverage" for each of the two preferences, while the two subspaces in (ii) only provide "partial coverage" for both preferences. Our weight definition captures this intuition with the use of $\|q_H\|^2$. Had we used $\|q_H\|$ instead, these subspaces would have identical weights.

If all preferences in $\mathcal{Q}_s$ lie within H, then $w(H) = |\mathcal{Q}_s| / (\dim(H))^\mu$, which is the maximum possible weight for subspaces with the same dimensionality. The term $(\dim(H))^\mu$ penalizes high-dimensional subspaces because constructing indexes for them are more expensive than for low-dimensional subspaces. The term also "normalizes" popularity, because a high-dimensional subspace is expected to be able to cover more preferences. By adjusting the parameter μ, we obtain a trade-off between keeping the indexing costs low and covering more preferences. Our experiments in Section VI use $\mu = \frac{1}{4}$.

Initializing candidate subspaces. For the purpose of finding core subspaces, we ignore insignificant attribute weights in preferences. Consider each preference $q \in \mathcal{Q}$. We round off any attribute weight to 0 if no greater than 0.01, and rescale the resulting preference so that it remains a unit vector.

Following this preprocessing, we say that a preference q is τ*-sparse* if $\dim(\mathsf{Sp}(q)) \leq \tau$ (i.e., q has non-zero weights for at most τ attributes). Since we are practically limited to core subspaces with dimensionality up to $\hat{\tau} = 5$, we focus on the subset $\mathbb{Q}_s$ of *sparse preferences*, i.e., those that are $(\hat{\tau} + \triangle\tau)$-sparse. Here, $\triangle\tau$ is a small slack (we set $\triangle\tau = 2$) that reflects the ability of our approach to handle denser preferences using multiple core subspaces.

We compute the set $\mathbb{K}$ of candidate core subspaces from the set $\mathbb{Q}_s$ of sparse preferences as follows. First, any $\hat{\tau}$-sparse preference gives us an axis-parallel candidate subspace: $\mathbb{K} \leftarrow \{\mathsf{Sp}(q) \mid q \in \mathbb{Q}_s \text{ and } q \text{ is } \hat{\tau}\text{-sparse}\}$. Second, for each sparse preference $q \in \mathbb{Q}_s$ that is not $\hat{\tau}$-sparse (but still $(\hat{\tau} + \triangle\tau)$-sparse), we consider all $\hat{\tau}$-dimensional axis-parallel subspaces of $\mathsf{Sp}(q)$ as candidates: $\mathbb{K} \leftarrow \mathbb{K} \cup \{\mathsf{Sp}(I) \mid \mathsf{Sp}(I) \subset \mathsf{Sp}(q) \text{ and } |I| = \hat{\tau}\}$.

If the size of $\mathbb{K}$ is small, we set $\mathbb{H}$ to $\mathbb{K}$ and stop, otherwise, we proceed to the next two stages. As mentioned in Section I, however, $\mathbb{K}$ can be large. For example, for $d = 20$, $\hat{\tau} = 5$, and $\triangle\tau = 2$, $|\mathbb{K}|$ can be as large as 21,699.

Adding popular subspaces. Suppose $\mathbb{K}$ has two overlapping subspaces of significant weights. It might be more efficient to build a single index for $\mathrm{span}(H_1, H_2)$ rather than building two separate indexes—one for H_1 and another for H_2. To enable this possibility, given $H_1, H_2 \in \mathbb{K}$, we add $H = \mathrm{span}(H_1, H_2)$ to $\mathbb{K}$ if all of the following conditions hold:

- $\dim(H) < \dim(H_1) + \dim(H_2)$; i.e., H_1 and H_2 overlap.
- $w(H_1)$, $w(H_2) \geq \mathrm{median}\{w(K) \mid K \in \mathbb{K}\}$, and $w(H) \geq 0.8(w(H_1) + w(H_2))$; i.e., H is sufficiently popular.
- $\dim(H) \leq \hat{\tau}$, where $\hat{\tau}$ is maximum dimensionality of a core subspace (introduced at the beginning of this section); we do not consider adding subspaces with higher dimensionality because indexing them would be too costly.

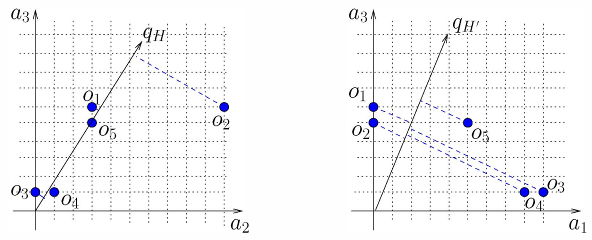
Selecting core subspaces. Continuing with the set $\mathbb{K}$ of candidate subspaces, we now compute a smaller set $\mathbb{H} \subseteq \mathbb{K}$, as *core subspaces*, to cover most of the sparse preferences in $\mathbb{Q}_s$. Note that we cannot simply choose the subspaces with the top weights because, together, they may overlap and end up covering only a small fraction of the preferences.

Algorithm 1 gives the pseudo-code of our approach. In each step, we select the subspace H with the highest weight out from $\mathbb{K}$. Importantly, every time we pick some H, we "update" the set of preferences in a way to reduce their contributions to subspace weights for those preferences covered by H. Thus, subsequent selections will focus on covering preferences that remain uncovered.

If a preference q is contained in H, q is fully covered by H. Otherwise, q is only partially covered. In this case, q_H, the projection of q on H, provides information about some of the attributes of q in the sense that the ranking of objects w.r.t. q_H gives some information about ranking of objects w.r.t. q—for those attributes that are present in H. We reduce the weights

Algorithm 1: SelectCoreSubspaces($\mathbb{K}$; δ).

```
$\mathbb{H} \leftarrow \emptyset$ ;
$m_s \leftarrow |\mathbb{Q}_s|$; remember the original value for each $q \in \mathbb{Q}_s$ (denoted $\tilde{q}$);
while $\frac{1}{m_s}\sum_{q \in \mathbb{Q}_s} \|q\| \geq \delta$ do
    foreach $K \in \mathbb{K}$ do compute $w(K)$ using Eq. (1);
    $H \leftarrow \arg\max_{K \in \mathbb{K}} w(K)$;
    $\mathbb{H} \leftarrow \mathbb{H} \cup \{H\}$; $\mathbb{K} \leftarrow \mathbb{K} \backslash \{H\}$;
    foreach $q \in \mathbb{Q}_s$ do
        $q \leftarrow q - \|\tilde{q}_H\| \cdot q_H$;
        if $\|q\| < \delta$ then $\mathbb{Q}_s \leftarrow \mathbb{Q}_s \setminus \{q\}$;
return $\mathbb{H}$;
```



(a) Subspace (a_2, a_3). (b) Subspace (a_1, a_3).

Fig. 2. Illustration of coverage. Here, $d = 3$, $k = 2$, $q = (0.2, 0.3, 0.5)$, and $\mathcal{O} = \{o_1, \ldots, o_5\}$, where $o_1 = (0, 3, 6)$, $o_2 = (0, 10, 5)$, $o_3 = (9, 0, 1)$, $o_4 = (8, 1, 1)$, and $o_5 = (5, 3, 5)$. Thus, $\pi_{\leq 2}\langle q, \mathcal{O}\rangle = \{o_2, o_5\}$. Suppose $H = (a_2, a_3)$ is selected. Then $\pi_{\leq 2}\langle q_H, \mathcal{O}_H\rangle = \{o_2, o_1\}$, as shown in Figure 2(a). If we simply clear any weights of attributes in H, q becomes $(0.2, 0, 0)$ and the top 2 projected objects w.r.t. attribute a_1 are o_3 and o_4. In this case, the correct second-ranked object o_5 will not be reported.

of those attributes in q that are present in q_H, so that subspaces we select in the future will capture the information of q w.r.t. the attributes of q not present in q. The simplest method will be to let $q \leftarrow q - q_H$; i.e., we simply clear q of any weights of attributes in H. However, this method is suboptimal; for a concrete example, see Figure 2.

Intuitively, for a partially covered preference, we would ideally like to cover each of its attributes with non-zero weights by multiple core subspaces. To this end, we update q using $q \leftarrow q - \|\tilde{q}_H\| \cdot q_H$, where $\tilde{q}$ denotes the original vector for the preference (while q denotes the current vector, whose value changes over the course of the algorithm). The multiplier $\|\tilde{q}_H\|$ ensures that if $\tilde{q}$ is partially covered by H (i.e., $\|\tilde{q}_H\| < 1$), we will leave some residual weights for attributes in H to encourage additional coverage. On the other hand, if $\tilde{q}$ is contained in H, the vector will become zero after the update, and there is no need to consider q further. Consider the same example in Figure 2. After H has been selected, q will become $(0.2, 0.06, 0.1)$. Suppose $H' = (a_1, a_3)$ is chosen. As shown in Figure 2(b), $\pi_{\leq 2}\langle q'_H, \mathcal{O}'_H\rangle = \{o_5, o_1\}$. Hence, the union of the top 2 objects in H and H', $\{o_1, o_2, o_5\}$, contains the exact top-2 objects, o_2 and o_5. In [17], we present experimental results that validate the effectiveness of multiple coverage.

In general, we stop covering a preference when its norm has dropped below a given significance threshold δ (e.g., 0.05). We stop selecting additional core subspaces altogether once the average norm of all preferences drops below δ; see [17] for additional evaluation on the choice of δ.

Discussion. As mentioned in Section I, our approach can be seen as a generalization of the view-based approach [5], [14]. The indexes we build for each core subspace H can be seen as a "super"-view that effectively provides the same power as materializing an infinite number of vector views whose vectors lie in H. On the other hand, unlike vector views, our core subspaces are axis-parallel. This restriction not only makes the problem more tractable, but also the attributes retain their meaning and if $\mathsf{Sp}(q)$ is a k-dim, then it will be k-dimensional even after the projection—number of non-zero attributes does not increase. It does not pose any issue for sparse preferences, because a multi-dimensional core subspace subsumes all vector views therein, including those that are not axis-parallel. Such degrees of freedom provided by multi-dimensional subspaces also make our approach more robust—while the choices of vector views are susceptible to errors and changes in the distributions of attribute weight values in preferences, our approach will still work well as long as preferences continue to specify non-zero weights, which can vary arbitrarily, for the same subsets of attributes.

IV. Constructing Indexes

We now describe the indexes we build. First, for each core subspace in $\mathbb{H}$, we build an object index for top-k queries (Section IV-A) and a preference index for reverse top-k queries (Section IV-B). The collection of these indexes for core subspaces aims at handling most (if not all) sparse preferences. Next, to handle all preferences not covered by these indexes, we separately build data structures for the full-dimensional space (Section IV-C).

For reverse top-k, in addition to these indexes, we also store the score of the k-th ranked object for each preference.

A. Core Subspace Indexes for Top-k Queries

For each core subspace $H \in \mathbb{H}$, a straightforward approach would to be project $\mathcal{O}$ onto H, and build an index on the $\dim(H)$-dimensional projected points that, given a query preference q, return the top k points with respect to q. This approach, however, has several issues. First, unless q is contained in H, there is a good chance that the we will miss some answers by looking only at the top k objects for q in H, even when we look in multiple core subspaces partially covering q. Second, indexing all points in $\mathcal{O}$ for every core subspace results would require $O(n|\mathbb{H}|)$ space, which is too much. Third, looking in multiple core subspaces per query means that the index for each core subspace must be fast.

To address these issues, for each core subspace H, we carefully choose a small subset of objects to build an index that supports top-βk queries in H. The index is small and fast, but approximate—a sensible trade-off because the top answers in a subspace in any case only approximate those in the full-dimensional space. Here, $\beta \geq 1$ is a small constant to increase the chance of catching a top-k object in the full-dimensional space. We set $\beta = 3$ in our experiments in Section VI; see [17] for additional evaluation on the choice of β.

Before going into more detail, we have to introduce the notion of *coreset*. For a preference $q \in \mathbb{S}^{d-1}$, let U_i and L_i

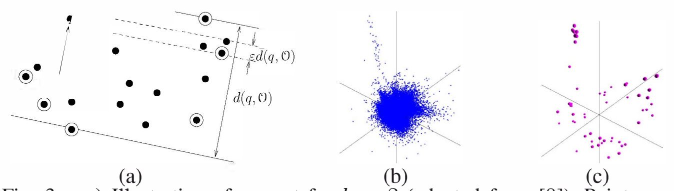


Fig. 3. **a)** Illustration of coreset for $k = 2$ (adapted from [8]). Points are shown as black dots and members of the coreset are circled. **b)** A set of objects generated for the *document subscription* workload (see Section VI). **c)** A coreset for the set of objects in (b); $k = 5$.

be the objects corresponding to the i-th maximum and the i-th minimum scores w.r.t. q. The difference between these two scores defines the ***i*-th *extent*** of $\mathcal{O}$ in direction q, denoted $d_i(q, \mathcal{O})$; i.e., $d_i(q, \mathcal{O}) = \langle q, U_i\rangle - \langle q, L_i\rangle$. Given an integer $k \geq 1$ and an error allowance 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 $q \in \mathbb{S}^{d-1}$,

$$\langle q, \pi_i(q, \mathcal{C})\rangle \geq \langle q, \pi_i(q, \mathcal{O})\rangle - \varepsilon\bar{d}_i(q, \mathcal{O}). \tag{2}$$

In other words, the i-th ranked object obtained by querying $\mathcal{C}$ in any direction is guaranteed to score closely as the actual i-th ranked object by querying $\mathcal{O}$ in the same direction.[1] Figure 3 illustrates the concept of coreset and shows an example for an object workload considered in Section VI. An algorithm for computing a (k, ε)-coreset of $\mathcal{O}$ of size $O(k/\varepsilon^{(d-1)/2})$ in $O(n \log n + k/\varepsilon^{3d/2})$ time was given in [16] (see also [8]).

Let $\mathcal{O}_H$ denote the projection of $\mathcal{O}$, the set of input objects, on H, and let $\varepsilon > 0$ be the error allowance. We construct an $(\beta k, \varepsilon)$-coreset of $\mathcal{C}_H \subseteq \mathcal{O}_H$ By definition, for any preference q in H, and for any $j \leq \beta k$, $\langle q, \pi_j(q, \mathcal{C}_H)\rangle \geq \langle q, \pi_j(q, \mathcal{O})\rangle - \varepsilon\bar{d}_j(q, \mathcal{O})$, i.e., the scores of top-βk objects of $\mathcal{C}_H$ are roughly the same as those of $\mathcal{O}$.

Next, we build an index on $\mathcal{C}_H$ such that for a query preference q in H and $\kappa \geq 1$, it returns $\pi_{\leq\kappa}(\mathcal{C}_H, q)$. By the definition of coreset, for $\kappa \leq \beta k$, the score of $\pi_{\leq\kappa}(\mathcal{C}_H, q)$ will be roughly the same as those of $\pi_{\leq\kappa}(\mathcal{O}_H, q)$. Many indexes are known for forward top-k queries; some provide provable bounds on their performance. Since this component is not the main focus of our work, our implementation simply uses $\dim(H)$ sorted lists on $\mathcal{C}_H$ and TA [13] for top-κ queries.

In the worst case, the total size of the index, summed over all core subspaces, is $O(\sum_{H\in\mathbb{H}} \beta k/\varepsilon^{(\dim(H)-1)/2})$. Since we cap the dimensionality of core subspaces at $\hat{\tau}$, the size is $O(\beta k|\mathbb{H}|/\varepsilon^{(\hat{\tau}-1)/2})$.

B. Core Subspace Indexes for Reverse Top-k

Let $\mathcal{Q}$ be the set of preferences with respect to which we wish to answer reverse top-k queries. On a high level, for each $q \in \mathcal{Q}$, we identify a small number of "covering core subspaces." Then, for each core subspace $H \in \mathbb{H}$, we index the subset of preferences that H covers. Before describing the indexes, we first discuss how to cover a preference.

[1] Note that $\varepsilon\bar{d}_i(q, \mathcal{O})$ provides a better error guarantee than $\varepsilon\langle q, \pi_i(q, \mathcal{O})\rangle$, because the former is independent of the choice of origin, preserved under affine transformation (e.g. translation, rotation, scaling), and smaller than the latter if all object attributes have non-negative values.

Algorithm 2: PreferenceCover($\mathbb{H}, q; \nu, \theta$)

```
1 $\Gamma \leftarrow \emptyset$, $\tilde{q} \leftarrow q$;
2 while $\|q\| \geq \theta$ and $|\Gamma| < \nu$ do
3     $H \leftarrow \arg\max_{H \in \mathbb{H}} \|q_H\|$;
4     if $\|q_H\| = 0$ then break;
5     $\Gamma \leftarrow \Gamma \cup \{H\}$; $\mathbb{H} \leftarrow \mathbb{H} \setminus \{H\}$;
6     $q \leftarrow q - \|\tilde{q}_H\| \cdot q_H$;
7 if $\|q\| \geq \theta$ then return $\emptyset$;
8 return $\Gamma$;
```

Covering a preference with core subspaces. A *cover* of a preference q, denoted Γ_q, is a subset of $\mathbb{H}$ onto which the projections of q are intended to preserve the information about q, in the sense described in Section III. A cover Γ_q is β-*perfect* with respect to $\mathcal{O}$ if for any query object $o \notin \mathcal{O}$ and $o \in \pi_{\leq k}(q, \mathcal{O} \cup \{o\})$, there exists a subspace $q \in \Gamma_q$ such that $o_H \in \pi_{\leq \beta k}(q_H, \mathcal{O}_H \cup \{o_H\})$. However, perfect covers are difficult to find. If q lies within a core subspace, then that subspace obviously is a 1-perfect cover of q. However, if none of the core subspaces contains q by itself, the best we can hope for is a small cover that preserves as much of q as possible.

A simple strategy would be to choose Γ_q to be those core subspaces that "overlap" with q (or more precisely, those on which q has a non-zero projection). However, there may be too many such subspaces; picking them all increases index space and slows down queries. We could pick the top subspaces based on the norms of q's projections on them, but doing so does not guarantee coverage of all non-zero attribute weights of q. Alternatively, we could pick the top subspaces according to their weights defined in Section III; however, weights are defined globally over $\mathcal{Q}$ and irrelevant for any particular q.

To avoid these problems, we set a limit ν on the maximum number of core subspaces in any cover, and use a greedy procedure (Algorithm 2) to cover q. The algorithm is similar to Algorithm 1 in spirit (though we are now covering only one q). In each step, we always pick the core subspace H for which q_H has the largest norm. More importantly, we update q for each step in a way that let subsequent picks focus on uncovered dimensions, while still encouraging multiple coverages for each dimensions (as discussed in Section III). This process is repeated until q is "mostly covered," i.e., the residual norm is less than a given threshold θ, or the cover size exceeds the limit ν. The choices of ν and θ allow the trade-off between coverage completeness and cost. We use $\nu = 3$ and $\theta = 0.5$ in our experiments in Section VI; see [17] for additional evaluation on their choices.

Note that some preferences may not be covered. Algorithm 2 returns $\emptyset$ if it cannot cover a preference. It is even possible (though not very likely) that some sparse preference cannot be covered. On the other hand, it is also possible to cover a non-sparse preference. Preferences that cannot be covered will be handled separately by data structures built in the full space $\mathbb{R}^d$ (Section IV-C). Our hope is that in practice, most preferences are sparse, can be covered, and will thus benefit from our approach.

Building the preference index. For each core subspace H, let $\mathcal{Q}^{(H)} = \{q_H \mid H \in \Gamma_q\}$ denote the subset of the preferences with H in their covers (as chosen by Algorithm 2). Our goal is to build an index that given $o \notin \mathcal{O}$, finds all preference $q \in \mathcal{Q}^{(H)}$ for which o_H ranks among the top βk objects in $\mathcal{O}_H \cup \{o_H\}$ for q_H. Assuming "near" β-perfect covers for all preferences, as discussed above, we know that if o enters the top-k answer of any preference q, then q will be returned by querying the preference index of some core subspace in Γ_q.

To build this preference index for $\mathcal{Q}^{(H)}$, we consider, for each preference $q \in \mathcal{Q}^{(H)}$, the score of the (βk)-th ranked object in $\mathcal{O}_H$ with respect to q_H, i.e., $\langle q_H, \pi_{\beta k}(q_H, \mathcal{O}_H)\rangle$. We call this score the *cutoff score*. Intuitively, we can determine whether a query object o_H enters the top-(βk) answer of q_H simply by comparing $\langle q_H, o_H\rangle$ with q_H's cutoff score. However, instead of working directly with $\mathcal{O}_H$, which is big, we work with $\mathcal{C}_H$, the $(\beta k, \varepsilon)$-coreset of $\mathcal{O}_H$ discussed in Section IV-A, which is much smaller. By definition, the score of the (βk)-th ranked object in $\mathcal{O}_H$ with respect to q_H is roughly the same as that of the (βk)-th ranked object in $\mathcal{C}_H$.[2]

Let $\mathcal{Q}_H = \{q_H \mid q \in \mathcal{Q}^{(H)}\}$ denote the projection of $\mathcal{Q}^{(H)}$ onto H. For a preference $q_H \in \mathcal{Q}_H$, let $\chi_q = \pi_{\beta k}(q_H, \mathcal{C}_H)$ be the cutoff score of q_H with respect to $\mathcal{C}_H$. Let r_q be the hyperplane $\langle x, q_H\rangle = \chi_q$ in H. By definition, the score of all objects $\mathcal{O}_H \in r_q$ is χ_q. Hence, for any object $z \in H \setminus \mathcal{C}_H$, $z \in \pi_{\leq \beta k}(q_H, \mathcal{C}_H \cup \{z\})$ iff $\langle z, q_H\rangle > \chi_q$, i.e. z lies in the *positive* halfspace $\langle x, q_H\rangle > \chi_q$ bounded by r_q. If we set $T_H = \{r_q \mid q \in \mathcal{Q}_H\}$, then for a query object z, we wish to report all hyperplanes of T_H such that z lies in their positive halfspaces. This is an instance of inverse halfspace range searching [18]. By using the so-called duality transform [19], T_H can be mapped to a set T_H^* of points in $\mathbb{R}^{\dim(H)}$ and the query reduces to reporting all points of T_H^* that lie in a query halfspace; see [8] and our technical report [17] for details. Several indexes for halfspace range searching are known. We simply use a kd-tree based index, similar to the one used in [8].

C. Handling Uncovered Preferences

As discussed in Section IV-B, core subspaces may not cover all preferences. To handle uncovered preferences, we build data structures in the full space $\mathbb{R}^d$.

For forward top-k queries by uncovered preferences, instead of working with the entire set of objects $\mathcal{O}$, we work with a coreset (just like in Section IV-A, but now in the full d-dimensional space). A (k, ε)-coreset of size $O(k/\varepsilon^{(d-1)/2})$ can be computed using the algorithm described in [16]. Because of the exponential dependence on d, the coreset can be large even for moderate values of d. While it is known that this size is required for the worst case [16], we show that if

[2]For a preference $q \in \mathcal{Q}^{(H)}$, the closer $\|q_H\|$ is to $\|q\|$, the more likely it is for an object highly ranked w.r.t. q_H to also rank high w.r.t. q. Thus, instead of defining the cutoff point using always the (βk)-th ranked object w.r.t. q_H, we can define it using the $(\beta' k)$-th ranked object, where $\beta' \in [1, \beta]$ is customized based on how close $\|q_H\|$ is to $\|q\|$. This heuristic expedites reverse top-k queries by tightening the cutoff condition; see our technical report [17] for more detailed discussion and evaluation (our experiments in Section VI does not employ this heuristic).

the input objects lie on a low-dimensional algebraic surface of constant degree, then we can compute a smaller coreset.

Theorem 1: Let $\mathcal{O}$ be a set of points in $\mathbb{R}^d$ that lie on a t-dimensional algebraic surface of constant degree, for $t < (d-1)/2$. Then, a (k, ε)-coreset of size $O((d^{3/2}/\varepsilon)^t)$ can be computed in time $d^{O(1)}n + O((d^{3/2}/\varepsilon)^t)$.

We omit the proof here due to space; it can be found in [17]. To answer a forward top-k query, we can simply scan the coreset of objects as long as the size of the coreset is not too large (which is the case for our workloads in Section VI); no additional indexing is needed. Otherwise, we can apply the TA-based approach at the end of Section IV-A.

For reverse top-k queries over uncovered preferences, we maintain the list of such preferences together with their k-th ranked object scores. We simply scan this list to answer a reverse top-k query. Because of the high-dimensional full space, more indexing is unlikely to bring significant benefits.

V. Query Procedure

Top-*k* query. Given a query preference $q \in \mathbb{S}^{d-1}$, we first call $\mathsf{PreferenceCover}(\mathbb{H}, q)$ (Algorithm 2) to compute Γ_q, a cover of q by core subspaces. There are two cases.

First, if $\Gamma_q = \emptyset$ (i.e., we cannot find a cover of q by $\mathbb{H}$), we query the coreset $\mathcal{C}$ of objects in the full space as described in Section IV-C with q and return $\pi_{\leq k}(q, \mathcal{C})$. Since $\mathcal{C}$ is a coreset of $\mathcal{O}$, the objects returned approximate $\pi_{\leq k}(q, \mathcal{O})$.

Otherwise, $|\Gamma_q| > 0$ and q is covered. For each $H \in \Gamma_q$, we compute q_H, the projection of q on H. We query the object index for H described in Section IV-A to obtain the set of objects $\mathcal{S}_H \in \mathcal{O}$ corresponding to $\pi_{\leq \kappa_H}(q_H, \mathcal{C}_H)$, where $\kappa_H = k$ if $\|q_H\| \approx 1$, or $\kappa_H = \beta k$ otherwise. Then, we compute $\pi_{\leq k}(q, \bigcup_{H \in \Gamma_q} \mathcal{S}_H)$, i.e., the top k objects among all returned objects, by calculating their actual scores w.r.t. q.

Reverse top-*k* query. Given a query object $o \in \mathbb{R}^d$, we want to report all affected preferences, i.e., any preference $q \in \mathcal{Q}$ for which o is a top-k object in $\mathcal{O} \cup \{o\}$ w.r.t. q. First, we find affected preferences among the uncovered preferences $\bar{\mathcal{Q}} \subseteq \mathcal{Q}$ as described in Section IV-C.

Next, we find affected preferences among the covered preferences, $\mathcal{Q} \setminus \bar{\mathcal{Q}}$. For each subspace $H \in \mathbb{H}$, we determine whether o is "relevant" to H, in the sense whether there can be some preference q in H for which o_H is potentially one of the top-βk objects of $\mathcal{C}_H \cup \{o_H\}$ w.r.t. q. The procedure for testing relevance is given in [8]; it takes $O(k/\varepsilon^{(d-1)/2})$ time in the worst case. If o is relevant to H, we query the preference index with o for H described in Section IV-B to find the affected preferences in H with their cutoff points. For each such preference q found, we further calculate o's actual score w.r.t. q in the full space, and return q only if o's score is higher than q's k-th score that we store (as discussed at the beginning of Section IV).

VI. Experimental Evaluation

Approaches compared. We compare our approach, hereafter called ***CSI*** (***C**ore-**S**ubspace-based **I**ndexing*), with a number of alternatives. All approaches are implemented in C^{++} and compiled by `g++` with options `-march=native` and `-O3`.

For top-k queries, we consider the following alternatives. ***Scan*** is a brute-force method that examines all objects.[3] ***BB*** indexes all objects in a d-dim kd-tree and uses a branch-and-bound algorithm to search for the top k objects. ***TA***, the Threshold Algorithm, keeps a list of objects sorted by each attribute; to find the top k objects give a preference q, it uses the lists for attributes with non-zero weights specified by q. ***PCA+TA*** first applies PCA (principal component analysis) to reduce the dimensionality of the objects, and then uses *TA*. ***Views***, the view-based approach, randomly selects as views a set of unit vectors from a given preference distribution, and materializes their top βk objects. Given a preference q, it retrieves the top βk objects from ν views most similar to q and computes the top k among these objects.

For reverse top-k queries, all approaches store the score of the k-th ranked object for each preference. ***Scan*** examines all preferences. ***HSR***, for *halfspace range search*, answers the query using a kd-tree as described in Section II. ***PCA+HSR*** first applies PCA and then uses *HSR* in the reduced space. ***Views*** selects views as described above, and assigns each preference to ν views; given a query object o, it retrieves all preferences assigned to views for which o enters their top-(βk) list, and filters these preferences to find those affected.

Since *CSI* is approximate, we set $\varepsilon = 0.08$ to be the error allowance, such that coresets are sized to provide answers whose scores are within ε times the directional width of the objects with respect to a preference (recall Eq. (2)). To ensure fair comparison between *CSI* and *Views*, we use the same settings of $\beta = 3$ and $\nu = 3$, and we choose the number of views such that the total space consumption of *Views* is the same as that of *CSI*.

Performance metrics. For a given query workload, we report the average wall-clock time per query over the workload, as measured on a Dell OptiPlex 990 with 3.40GHz Intel Core i7-2600 CPU, 8MB cache, and 8GB memory.

For approximate approaches to top-k queries (*CSI*, *PCA+TA*, and *Views*), we measure the approximation error for each query object o as follows. Let $\tilde{o}_i$ denote the i-th ranked object returned by an algorithm. The error is computed as $\max_{i\in[1,k]} \frac{\langle q, \pi_i(q,\mathcal{O})\rangle - \langle q, \tilde{o}_i\rangle}{\varepsilon \bar{d}_i(q,\mathcal{O})}$, where ε is the error allowance as set above. Thus, an error of 1 or less is considered "acceptable." We report the RMS (root mean square) error over the query workload. If RMS error is 1 or higher, it is likely that a significant fraction of the errors are unacceptable.

[3] *Scan* computes the score for every object, while maintaining a buffer for the top k objects seen so far. The objects are stored simply in arrays. We note that additional performance improvements may be possible, e.g., by storing the objects by attributes (i.e., in a columnar format) to improve cache performance. However, these improvement will also benefit *CSI*, because it uses *Scan* for uncovered preferences (Section IV-C). For example, in Figure 6, where 90% of the queries are covered for 500 subspaces, the query time for *CSI* is $0.9(\mathit{CSI}^*$ time$) + 0.1(\mathit{scan}$ time$) = 0.9 \times 0.05 + 0.1 \times 7.5 = 0.79$ms. If *scan* time improves to 1ms, *CSI*'s will improve to 0.145ms. Furthermore, *CSI* is flexible in processing top-k queries in core subspaces: we can use *Scan* instead of TA (Section IV-A) if a highly-tuned *Scan* can beat TA.

For approximate approaches to reverse top-k queries (*CSI*, *PCA+HSR*, and *Views*), we measure their approximation qualities using *false negative rates* defined as follows. Given a query object o, a preference q is considered to be *significantly* affected by o iff $\langle q, o\rangle > \langle q, \pi_k(q, \mathcal{O})\rangle + \varepsilon \bar{d}_k(q, \mathcal{O})$; here, the same ε we set earlier defines the amount of acceptable slack. If a significantly affected q is missing from the query result, we count it as a false negative. We divide the total number of false negatives by the total actual number of significantly affected preferences over the entire query workload, and report this ratio as the false negative rate.

Synthetic object workloads. We generate objects using a number of distributions. With ***box-uniform***, objects are distributed uniformly and randomly *within* the unit box in $\mathbb{R}^d$. With ***sphere-uniform***, objects are distributed uniformly and randomly *on the surface of* the unit sphere in $\mathbb{R}^d$. With ***sector-select***, objects are drawn randomly from a spherical cap in $\mathbb{R}^d$ with apex at the origin, and with radius 1 and cone angle $15°$; furthermore, we only generate an object if it ranks high w.r.t. some preference in the preference workload. With ***t-surface***, objects lie on a t-dimensional algebraic surface embedded in the ambient space and represented in a parametric form.

Synthetic preference workloads. Our preference workload generator uses a number of parameters to control workload characteristics. Given a ***fraction of non-sparse preferences***, we generate this fraction of the preferences in the workload by picking unit vectors in $\mathbb{R}^d$ uniformly at random; assuming a sufficiently large d, such preferences are almost always non-sparse. For the remaining (sparse) preferences, we generate them from a set $\mathbb{G}$ of "generating subspaces," where $|\mathbb{G}| = h_{\text{gen}}$, the ***number of generating subspaces***, and for each $G \in \mathbb{G}$, $\dim(G) \leq \tau_{\text{gen}}$, the ***maximum generating density***. We pick $\mathbb{G}$ in two ways: with ***uniform generating subspaces***, every subspace with dimensionality no more than τ_{gen} has an equal probability of being picked; with ***skewed generating subspaces***, we assign each attribute a popularity, such that popular attributes are more likely to be included in a generating subspace. To generate a preference, we select a generating subspace $G \in \mathbb{G}$ at random. Then, we generate the preference in two ways: with ***uniform preferences within subspaces***, we draw a unit vector in G uniformly at random; with ***clustered preferences within subspaces***, we draw preferences from a mixture distribution centered around a small number of randomly chosen unit vectors in G.

NBA workload. This dataset contains 17 career stats for 3,861 NBA players. Preferences are generated synthetically.

Document subscription workload. This workload is intended to approximate an application scenario where users subscribe to documents of their interest. We obtain the set of objects representing documents from the collection of approximately 300,000 NY Times news articles [20]. We perform a singular value decomposition (SVD) on the documents to discover the underlying 20 most relevant topics. Hence, each document is mapped to a point in the 20-dimensional space, where each attribute represents a topic.

Next, we use the Yahoo! search query collection [21] to extract the set of preferences for this workload. This collection contains a random sample of 4,496 queries posted to Yahoo!'s US search engine in January, 2009. We preprocess the queries to discard stop words and words that are not present in the document collection. Then, using the same SVD matrices, we map each query to a unit vector in the 20-dimensional space; if a component of the vector is below a threshold t, we set it to 0. The table below shows, for two different t values, the density (number of non-zero components) distribution of resulting vectors (recall that $d = 20$):

density	0	1	2	3	4	5	6	7	8	9	10	11	12
# vectors ($t = 0.05$)	1558	0	0	3	32	113	342	749	864	567	209	51	8
# vectors ($t = 0.1$)	1592	338	1010	1084	390	80	2	0	0	0	0	0	0

We set $t = 0.1$. To get more preferences, we generate them from the above set of "seed" vectors. We associate each word with its 5 most "probable" topics (derived from the same SVD). Two words are neighbors if they are associated with a common topic. Starting from a seed vector, we generate new preferences by iteratively replacing a word with a neighbor.

A. Top-k Query Performance

This section will show that 1) *CSI* is at least an order of magnitude faster than the exact approaches and significantly more accurate than *PCA*; 2) although *CSI* and *Views* have comparable query times in most cases, *CSI* gives more accurate answers and works better in higher dimensions, across preference distributions, etc.; 3) *CSI*'s query times for covered preferences are at least an order of magnitude shorter than *Views*; and 4) *CSI* need not know the weight distribution of preferences to exploit their sparsity, and hence is less susceptible to changes in preference distribution than *Views*. We will also briefly comment on *CSI*'s index size and construction time.

Varying the fraction of non-sparse preferences. We begin by studying the effect of the fraction of non-sparse preferences on top-k queries for various approaches. Here, $d = 80$, $k = 5$, and we use 100,000 objects from *box-uniform*. The query workload consists of 10,000 preferences; the sparse ones among them are *uniform preferences* drawn from 200 *skewed generating subspaces* with maximum dimensionality $\tau_{\text{gen}} = 6$. *CSI* and *Views* are given 10,000 preferences generated from the same distribution in constructing their indexes. In Figure 4, we vary the fraction of non-sparse preferences from 0.02 to 0.64. For *CSI*, the RMS error is significantly below 1 at all times, but the overall average query time rises with more non-sparse preferences. The table below shows the fraction of preferences covered by core subspaces, which has a roughly linear relationship with the fraction of sparse preferences:

Fraction of non-sparse preferences	0.02	0.04	0.08	0.16	0.32	0.64
Fraction of covered queries	92.3%	90.6%	86.8%	79.1%	63.4%	30.8%

Recall that *CSI* uses indexes in core subspaces for covered preferences, and the full-dimensional coreset for uncovered

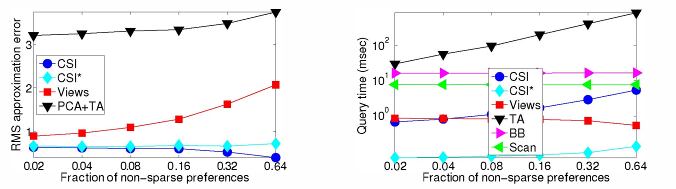


Fig. 4. Top-k queries when varying the fraction of non-sparse preferences; $d = 80$, $n = 100{,}000$, $k = 5$.

preferences. To better see their performance difference, in this and the following figures, we show the average query time and RMS error for covered preferences using the notation *CSI**. When most preferences are non-sparse, they are handled by the full-dimensional coreset, so *CSI* becomes as slow as *scan* and *BB*,[4] which is expected in high dimensions. This observation implies that using only the full-dimensional coreset (as well as other full-dimensional approaches such as the layer-based ones mentioned in Section I) will not work in high dimensions.

The error of *Views* is acceptable when all preference are sparse. However, its error quickly deteriorates as the fraction of non-sparse preference rises, because of the inherent difficulty in capturing high-dimensional space with vector-based views. Although the query-time plot shows an apparent advantage of *Views* over *CSI*, this advantage is unattainable in practice—to make its error acceptable, *Views* would have to use a lot more views, driving the space and query time higher than *CSI*.

Figure 4 shows that *PCA+TA* does not produce acceptable errors; thus, we do not plot its query time. Also, we do not plot error for *scan*, *BB*, and *TA* because they are exact methods.

Now that the effect of non-sparse preferences is well understood, we will focus on workloads where all preferences are sparse—extrapolation to the general case is easy, and *Views* will only be worse than *CSI* with more non-sparse preferences.

Varying preference workloads. We now examine several different preference workloads. In Figure 5, $d = 20$, and preferences (either for querying or for index construction) are generated from *uniform generating subspaces*; in Figure 6, $d = 80$, and generating subspaces are *skewed*. For both figures, we vary the number of generating subspaces. Other workload parameters remain the same as Figure 4. The main observation is that the exact methods run much slower than the approximate ones (note the logarithmic scale of the query time axis). *CSI* and *Views* and have comparable query time, but *CSI* has smaller errors than *Views*. *PCA+TA* again produces much higher errors than *CSI* and *Views*. We observe similar trends for increasing k; because of limited space, we report experimental results on varying k in [17].

CSI object index size and construction time. Coresets for core subspaces are small—a few hundred objects per core subspace—because their size depends on error and dimensionality, not the total number of objects. For example, for Figure 5 (16MB data), *CSI* object index size ranges from 100KB to 130KB and the construction time ranges from 30s to 50s for all tested cases. For Figure 6 (64 MB data), *CSI* object index size ranges from 160KB to 580KB and the construction time ranges from 1 to 4 minutes. Such construction times are acceptable because they are incurred once and amortized over the entire query workload. Also, our implementation of index construction is rather basic; improvement is possible with a good kNN library.

[4]*TA* is slower with more non-sparse preferences, because each such preference requires processing d lists and is thus more costly than a sparse one.

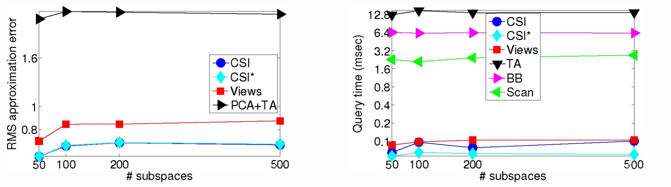


Fig. 5. Top-k queries when varying the number of *uniform* generating subspaces; $d = 20$, $n = 100{,}000$, $k = 5$.

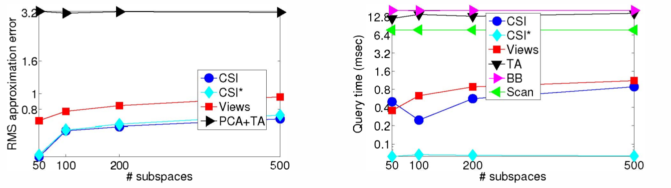


Fig. 6. Top-k queries when varying the number of *skewed* generating subspaces; $d = 80$, $n = 100{,}000$, $k = 5$.

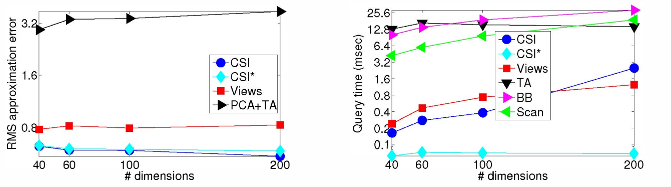


Fig. 7. Top-k queries when varying d; $n = 100{,}000$, $k = 5$.

Varying dimensionality. Going from Figure 5 to Figure 6, queries generally become slower with a higher dimensionality, but as indicated by *CSI**, query times for covered preferences remain short, and become much shorter than *Views*. As majority of the queries are covered, they will benefit from shorter-than-average query times. On the other hand, the accuracy lead of *CSI* over *Views* is consistent in both Figures 5 and 6.

We further show the impact of dimensionality in Figure 7. In this setting, $k = 5$, and we use 100,000 objects from *box-uniform*. Preferences are drawn as *uniform preferences* from 100 *uniform generating subspaces* with maximum dimensionality $\tau_{\text{gen}} = 6$. We see that *CSI* consistently delivers higher accuracy than *Views* across all dimensionalities, and its big lead over *PCA+TA* widens as d increases. While *Views* starts out to be slightly faster than *CSI* in low dimensions, the speed gap between them quickly narrows in higher dimensions. The exact methods are generally much slower *CSI* and *Views*. Finally, looking at *CSI**, we see that covered queries remain extremely fast despite the increase in d, meaning that core subspaces do a good job of protecting sparse preference query performance from the curse of dimensionality.

Objects from low-dimensional algebraic surfaces. In this experiment, we draw a varying number of objects from *t-surface* (a 3-dimensional bounded-degree algebraic surface to

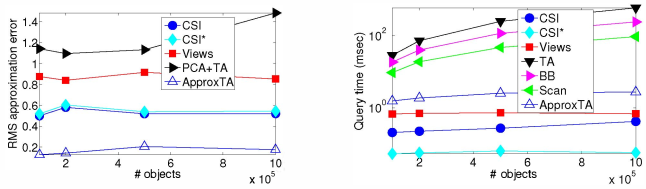


Fig. 8. Top-k queries for objects from *t-surface*; $d = 100$, $k = 5$.

be specific). Here, $d = 100$, and preference workloads are generated by drawing 10,000 *uniform preferences* from 100 *uniform generating subspaces* with maximum dimensionality $\tau_{\text{gen}} = 6$. In Figure 8, we see that *CSI*'s query time (which accounts for uncovered query preferences that will use the full-dimensional coreset) increases at a very slow rate as the number of objects increases. In fact, despite high dimensionality ($d = 100$), the size of *CSI*'s full-dimensional coreset is only around 8,000 even when $n = 1{,}000{,}000$, confirming the effectiveness of our improvement to the coreset construction algorithm discussed in Section IV-C. In comparison, the exact methods are much slower, and the gap widens as n increases. *Views* is also slower than *CSI*, but its query time remains steady thanks to *CSI*'s small coreset size (recall that we set the space of *Views* to be the same as that of *CSI*).

Figure 8 also shows an approximate variant of *TA* called *ApproxTA*, which simply runs *TA* on the full-dimensional coreset used by *CSI*, for all query preferences. Between *ApproxTA* and *CSI*, there is a clear tradeoff—*ApproxTA* has better accuracy, while *CSI* has faster speed. This comparison highlights the benefit of our improved coreset construction algorithm, as well as the ability for core subspace to further provide good accuracy/speed trade-offs.

Sensitivity to changes in preference distribution. In Section IV-A, we argued that *CSI* is more robust than *Views* with respect to errors and changes in the distributions of attribute weight values. We now validate this claim using the following experiment. Here, $d = 80$, $k = 5$, and we use 100,000 objects from *sector-select*. We define two preference workload distributions W_1 and W_2. For both, we use *uniform generating subspaces* with maximum dimensionality $\tau_{\text{gen}} = 3$; we also use these subspaces to generate the sectors for *sector-select* objects. W_1 and W_2 both draw *clustered preferences* from each generating subspace, but they have different set of cluster centers. To construct their indexes, we give *CSI* and *Views* 10,000 preferences from W_1. Then, we compare the performance of *CSI* and *Views* when given 10,000 preferences from W_1 (i.e., preference distribution is *unchanged*) and when given query 10,000 preferences from W_2 (i.e., preference distribution is *changed*).

Figure 9 plots the results as we vary the number of generating subspaces; results for which the preference distribution is unchanged are shown as "baseline." We see that while *Views* has a very accurate baseline (because the preferences are highly clustered), its accuracy becomes unacceptable when the preference distribution changes. In contrast, *CSI* remains highly accurate despite the change.

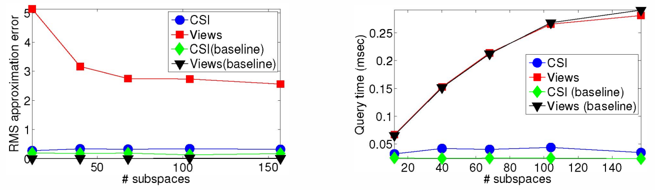


Fig. 9. Sensitivity of top-k query performance to changes in preference distribution within generating subspaces; $d = 80$, $n = 100{,}000$, $k = 5$.

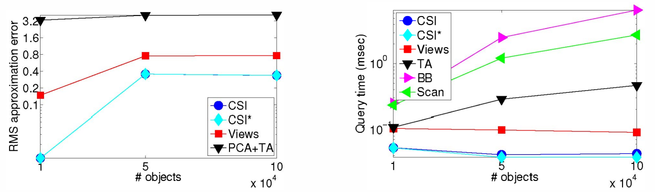


Fig. 10. Top-k queries for document subscription workload; $d = 20$, $k = 5$.

Document subscription workload. Figures 10 shows the results for the document subscription workload when varying the number of documents. Once again, the results confirm the effectiveness of *CSI*. Almost 100% of the preferences can be handled by core subspaces, and the average query time is much faster than the exact methods and comparable with *Views*. In comparison, *Views* has bigger approximation errors, and *PCA+TA* is worse. In fact, under *CSI*, at most 2% of the queries exceed the prescribed error allowance (i.e., approximation error is greater than 1). In contrast, up to 8% and 84% of preferences have approximation errors greater than 1 under *Views* and *PCA+TA*, respectively.

B. *Reverse Top-k Query Performance*

This section will show that 1) *CSI* is significantly faster than *Scan* and *HSR*, and 2) its false negative rate is much lower than *PCA* and *Views*, and never exceeds 10% in all tested cases.

Varying dimensionality. We begin by studying the effect of dimensionality on reverse top-k queries for various approaches. Here, $k = 5$. We draw 10,000 objects from *box-uniform*, and 100,000 *uniform preferences* from 100 *uniform generating subspaces* with maximum dimensionality $\tau_{\text{gen}} = 6$. Query objects are also drawn from *box-uniform*. Figure 11 shows the results. As with top-k queries, we see a similar pattern in accuracy: *CSI* misses very few significantly affected preferences (4% to 6%); *Views* misses 21% to 70% as d increases; *PCA+HSR* misses over 98%. In terms of query time, *HSR* is the slowest because in high dimensions, a query halfspace intersects more nodes of the underlying kd-tree, and the cost of determining whether a cutoff point lies above a hyperplane grows proportionally. Thus, *HSR* turns out to be even slower than *Scan*. Among the approximate methods, both *PCA+HSR* and *Views* are faster than *CSI*, but as we have seen, they have poor accuracy. *CSI* is able to maintain high accuracy and offer a significant speedup over *Scan* even at $d = 200$.

Recall that for each query, *CSI* also checks the full-dimensional index of uncovered preferences, basically using *Scan*. This cost component is reflected in the query times we report, and depends on the fraction of the uncovered preferences. In the worst case, if all preferences are non-sparse,

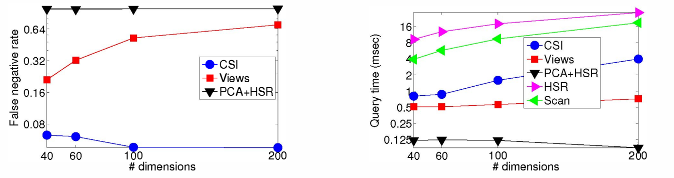


Fig. 11. Reverse top-k queries when varying d; $n = 10{,}000$, $m = 10{,}000$, $k = 5$.

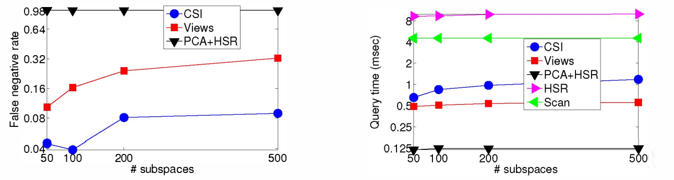


Fig. 12. Reverse top-k queries when varying the number of generating subspaces; $d = 40$, $n = 10{,}000$, $m = 100{,}000$, $k = 5$.

many of them will not be covered, and the query time of *CSI* will be similar to that of *Scan*. Because it is easy to extrapolate the effect of varying the fraction of non-sparse preferences, we set this fraction to 0 and do not vary it for the synthetic workloads in this section.

Varying the number of generating subspaces. We use the same workload parameters as in Figure 11, but vary the number of generating subspaces while fixing $d = 40$. Figure 12 shows the results. We see a similar trade-off as in Figure 11: *Views* and *PCA+HSR* run faster than *CSI*, but offer much lower accuracy; the exact methods are much slower.

We see that the number of generating subspaces affects *CSI*. More generating subspaces imply more diversity in preferences, which leads to more core subspaces, as well as a larger number of imperfectly covered preferences. Hence, both false positive rate and query time increase, although the effect is not strong enough to change any conclusion above.

Varying the number of preferences. Next, we study the effect of the number of preferences. We use the same preference workload parameters as in Figure 11, but vary the number of preferences up to 500,000. This time, the 10,000 objects are from *sphere-uniform*, and we draw query objects also from *sphere-uniform*. From Figure 13, we see that the same trade-off identified in previous figures continues: *CSI* is slower than *Views* and *PCA+HSR*, but is more accurate. The exact methods are much slower, while the fastest approximate method, *PCA+HSR*, misses most of the answers.

Overall, *CSI* demonstrates good scalability in the number of preferences. With half a million preferences, *CSI*'s false negative rate is merely 2%, and average query time is under 5.2 milliseconds. A more detailed breakdown shows that it spends 1.34ms querying indexes for core subspaces (454,270 out of 500,000 preferences), and 1.96ms filtering false positives; it also spends 1.9ms on checking the full-dimensional index for the remaining uncovered preferences. In comparison, the average query time of *Views* is about 3.24 milliseconds, 88% of which is spent on filtering false positives.

NBA workload. Figures 14 compares various approaches for the NBA workload as we increase the number of preferences.

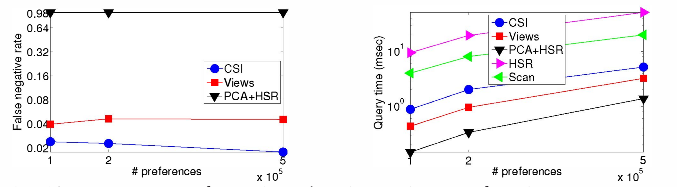


Fig. 13. Reverse top-k queries when increasing m; $d = 40$, $n = 10{,}000$, $k = 5$.

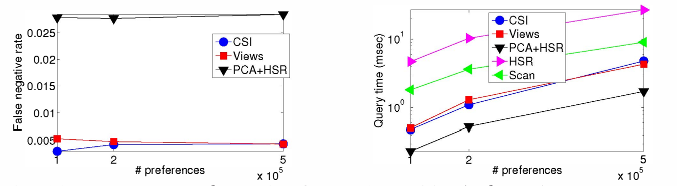


Fig. 14. Reverse top-k queries for NBA workload; $d = 17$, $n = 3{,}861$, $k = 5$.

We draw *uniform preferences* from 100 *uniform generating subspaces* with maximum dimensionality $\tau_{\text{gen}} = 6$. To ensure that the query objects are "interesting" (i.e., likely affecting some preferences), we test the reverse top-k queries using Hall-of-Fame players as query objects. In Figure 14, we see that *CSI* and *Views* both achieve low false negative rate among all approximate methods, while still delivering fast query time with a large number of preferences.

Document subscription workload. For this workload, we use 10,000 documents and vary the number of preferences up to 500,000. Figure 15 shows the results. As with the NBA workload, both *Views* and *CSI* perform well. *CSI* offers a nice middle ground between *HSR* and *Views*: on one hand, *CSI* is roughly 4 (10) times faster than faster than *Scan* (*HSR*); on the other hand, it is slightly slower than *Views*, but its false negative rate is 7 times fewer than that of *Views* at $m = 500{,}000$. For *CSI*, the false negatives rate is less than 0.004% across all tested workloads.

VII. Related Work

Preference top-k and reverse top-k queries. As already discussed in Section I, there has been a lot of work on preference top-k queries [1], [2], [3], [4], [5], [6], [7], [9] and reverse top-k queries [10], [8]. We build on and compare with our previous work in [8], which applied the ideas of coreset and duality transform to the full-dimensional space; this reference also provides additional discussion of and comparison with other previous approaches to top-k and reverse top-k queries. Another recent work on reverse top-k [11] uses a branch-and-bound algorithm on an R-tree, but as mentioned in Section I, its scalablility is limited to low dimensions.

The layer-based approaches (e.g., [1]) are essentially the exact counterpart of coresets, and are subsumed by coresets because the latter provides more flexible accuracy/space trade-offs. Thus, we do not compare directly with the layer-based approach or the hybrid approach [7], [9] that builds on them; their difficulty with high dimensions can be seen from the performance gap between *CSI* and *CSI** in Section VI-A.

Top-k queries can be seen as a special case of rank aggregation [22], and the Threshold Algorithm [13] is a viable

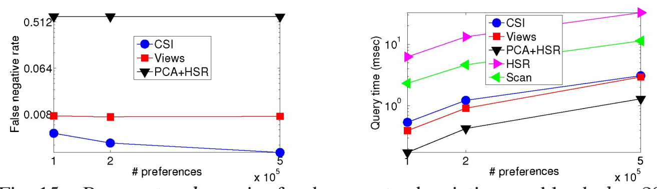


Fig. 15. Reverse top-k queries for document subscription workload; $d = 20$, $n = 10{,}000$, $k = 5$.

option for top-k queries. We compare with *TA* extensively in Section VI-A.

Finding interesting subspaces. Our task of identifying core subspace is related to the problems of *subspace clustering* (finding all clusters in all subspaces) and *projective clustering* (assigning points to clusters that exist in different subspaces). There has been a lot of work on these problems (see [23] for a survey). In particular, if the subspaces are axis-parallel, the problem is also related to the so-called *row/column-subset selection* problem [24], [25]: given a matrix where rows are objects and columns are features, select a subset of features that are dominant. However, the intended use of our core subspaces warrants our specialized algorithm in Section III, which accounts for the feature of multiple and partial coverage for a preference.

While we choose to make our core subspaces axis-parallel for reasons of simplicity and robustness against changes in attribute weight distributions, there are some situations for which it may be beneficial to consider subspaces that are arbitrarily oriented. For example, the preference workload may be known and stable. As another example, preferences may not exhibit sparsity in the original space, but do so after some affine transformation. In these situations, the problem of finding arbitrarily oriented subspaces is related to *subspace segmentation*, which seeks to model a set of data points using a union of affine subspaces (see [26] for a survey). PCA can be seen as a very restrictive special case where all points come from a single affine subspace; as we have seen in Section VI, it is less effective than multiple axis-parallel subspaces. Considering multiple arbitrary core subspaces in our solution remains an interesting problem for future work.

VIII. Conclusion

In this paper, we proposed a solution, based on the idea of core subspaces, for top-k and reverse top-k queries in high dimensions. Our solution exploits the sparsity in preferences to identify core subspaces, and applies the techniques of coresets and duality transform to index each core subspace as well as the full-dimensional space effectively. As shown by our experimental evaluation, in high dimensions, exact methods are slow, while existing approximation methods suffer from either poor speed (e.g., when using only a single coreset in the full space) or poor accuracy (such as the PCA- and view-based approaches). In contrast, for workloads where preferences are often sparse—a case that we believe arises naturally in practice—our solution offers a desirable trade-off between speed and accuracy, which makes scalable processing of top-k and reverse top-k queries in high dimensions a reality.

Acknowledgment

The authors have been supported by NSF under grants CCF-09-40671, CCF-10-12254, CCF-11-61359, IIS-09-16027, and IIS-1320357; by ARO under grants W911NF-07-1-0376 and W911NF-08-1-0452; and by an ERDC contract W9132V-11-C-0003.

References

[1] 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 2000*, pp. 391–402.
[2] V. Hristidis, N. Koudas, and Y. Papakonstantinou, "Prefer: a system for the efficient execution of multi-parametric ranked queries," *SIGMOD 2001*, pp. 259–270.
[3] P. Tsaparas, N. Koudas, Y. Kotidis, T. Palpanas, and D. Srivastava, "Ranked join indices," *ICDE 2003*, pp. 277–288.
[4] K. Mouratidis, S. Bakiras, and D. Papadias, "Continuous monitoring of top-k queries over sliding windows," *SIGMOD 2006*, pp. 635–646.
[5] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis, "Answering top-k queries using views," *VLDB 2006*, pp. 451–462.
[6] G. Das, D. Gunopulos, N. Koudas, N. Sarkas, "Ad-hoc top-k query answering for data streams," *VLDB 2007*, pp. 183–194.
[7] J.-S. Heo, J. Cho, and K.-Y. Whang, "The hybrid-layer index: A synergic approach to answering top-k queries in arbitrary subspaces," *ICDE 2010*, pp. 445–448.
[8] A. Yu, P. K. Agarwal, and J. Yang, "Processing a large number of continuous preference top-k queries," *SIGMOD 2012*, pp. 397–408.
[9] J.-S. Heo, J. Cho, and K.-Y. Whang, "Subspace top-k query processing using the hybrid-layer index with a tight bound," *Data Knowl. Eng.*, 83:1–19, 2013.
[10] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Norvag, "Reverse top-k queries," *ICDE 2010*, pp. 365–376.
[11] A. Vlachou, C. Doulkeridis, K. Nørvåg, and Y. Kotidis, "Branch-and-bound algorithm for reverse top-k queries," *SIGMOD 2013*, pp. 481–492.
[12] J. Matoušek, "Reporting points in halfspaces," *Comput. Geom. Theory Appl.*, 2:169–186, 1992.
[13] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *PODS 2001*, pp. 102–113.
[14] V. Hristidis and Y. Papakonstantinou, "Algorithms and applications for answering ranked queries using ranked views," *The VLDB Journal*, 13(1):49–70, 2004.
[15] W. B. Johnson and J. Lindenstrauss, "Extensions of Lipschitz mappings into a Hilbert space," *Contemporary Mathematics*, 26:189–206, 1984.
[16] P. K. Agarwal, S. Har-Peled, and H. Yu, "Robust shape fitting via peeling and grating coresets," *Discrete & Computational Geometry*, 39(1–3):38–58, 2008.
[17] A. Yu, P. K. Agarwal, and J. Yang, "Top-k preferences in high dimensions," Duke University, Tech. Rep., 2013. http://www.cs.duke.edu/dbgroup/papers/TR13-YuAgarwalYang-topk_pref_high_dim.pdf
[18] P. K. Agarwal and J. Erickson, "Geometric range searching and its relatives," *Contemporary Mathematics*, 223:1–56, 1999.
[19] J. Matousek, *Lectures on Discrete Geometry*. Springer-Verlag, 2002.
[20] A. Asuncion and D. Newman, "UCI machine learning repository," 2007. http://www.ics.uci.edu/~mlearn/MLRepository.html
[21] Yahoo! Search Query Tiny Sample. http://webscope.sandbox.yahoo.com/catalog.php?datatype=l
[22] R. Fagin, "Combining fuzzy information from multiple systems (extended abstract)," *PODS 1996*, pp. 216–226.
[23] H.-P. Kriegel, P. Kröger, and A. Zimek, "Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering," *ACM Trans. Knowl. Discov. Data*, 3(1):1–58, 2009.
[24] A. Deshpande and L. Rademacher, "Efficient volume sampling for row/column subset selection," *FOCS 2010*, pp. 329–338.
[25] V. Guruswami and A. K. Sinop, "Optimal column-based low-rank matrix reconstruction," *SODA 2012*, pp. 1207–1214.
[26] A. Aldroubi, "A review of subspace segmentation: Problem, nonlinear approximations, and applications," *Signal Processing Review*, 2012.