Towards Robust Indexing for Ranked Queries *

Dong Xin Chen Chen Jiawei Han
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, 61801
{dongxin, cchen37, hanj}@uiuc.edu

ABSTRACT

Top-k query asks for k tuples ordered according to a specific ranking function that combines the values from multiple participating attributes. The combined score function is usually linear. To efficiently answer top-k queries, preprocessing and indexing the data have been used to speed up the run time performance. Many indexing methods allow the online query algorithms progressively retrieve the data and stop at a certain point. However, in many cases, the number of data accesses is sensitive to the query parameters (*i.e.*, linear weights in the score functions).

In this paper, we study the sequentially layered indexing problem where tuples are put into multiple consecutive layers and any top-k query can be answered by at most k layers of tuples. We propose a new criterion for building the layered index. A layered index is *robust* if for any k, the number of tuples in the top k layers is minimal in comparison with all the other alternatives. The robust index guarantees the worst case performance for arbitrary query parameters. We derive a necessary and sufficient condition for robust index. The problem is shown solvable within $O(n^d \log n)$ (where d is the number of dimensions, and n is the number of tuples). To reduce the high complexity of the exact solution, we develop an approximate approach, which has time complexity $O(2^d n(\log n)^{r(d)-1})$, where $r(d) = \lceil \frac{d}{2} \rceil + \lfloor \frac{d}{2} \rfloor \lceil \frac{d}{2} \rceil$. Our experimental results show that our proposed method outperforms the best known previous methods.

1. INTRODUCTION

Rank-aware query processing is important in database systems. The answer to a top-k query returns k tuples ordered according to a specific score function that combines the values from participating attributes. In many cases, the combined score function is linear, while the weights in the linear ranking functions may vary dramatically with different users. One example is college ranking [1]. Every year *US News and World Report* ranks school performance by a linear weighting of different factors such as research quality assessment, tuition and fee, graduate employment rate, *etc.*. To search for the best schools with respect to each individual preference, students will generate their own ranking by assigning different weights. For example, students with budget concern may put a high weight on "tuition and fee", while students looking for good future employment will put a high weight on "graduate employment rate". For another example, consider a database containing houses available for sale [6]. Each house has several attributes, such as price, distance to the school nearby, floor area, *etc.*. Different users may also come up with different weighting strategies.

Database system should be able to process the ranked queries efficiently with respect to *ad hoc* linear weights. Since users usually have fixed positive (or negative) preferences on the attributes, we further assume that the linear weighting function is monotone (*i.e.*, all weights are non-negative). The extension to non-monotone functions will be addressed later in this paper. Without loss of generality, we assume that *minimization* queries are issued in this paper. A naïve method to answer such a top-k query is to first calculate the score of each tuple, and then output the top-k tuples from them. This approach is undesirable for querying a relatively small value of k from a large data set. Pre-processing and indexing the data have been used to speed up run time performance. Particularly, we are interested in sequential indexing approach for the following two reasons. First, it can be easily integrated into a database system without sophisticated data structures or query algorithms; and second, it enables sequential access of data which may reduce database I/Os. The sequential indexing approach projects multi-dimensional data points onto a one-dimensional index. The index can be either layered or not layered.

Recent successful work in non-layered approaches includes the PREFER[1] system [13], where tuples are sorted by a pre-computed linear weighting configuration. Queries with different weights will be first mapped to the pre-computed order and then answered by determining the lower bound value on that order. When the query weights are close to

*The work was supported in part by the U.S. National Science Foundation NSF IIS-03-08215/05-13678 and NSF BDI-05-15813. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

VLDB '06, September 12-15, 2006, Seoul, Korea.

[1]The original PREFER system is based on views, here we borrow the idea to build the index.

the pre-computed weights, the query can be answered extremely fast. Unfortunately, this method is very sensitive to weighting parameters. A reasonable derivation of the query weights (from the pre-computed weights) may severely deteriorate the query performance (as shown in example 1).

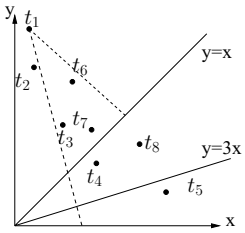

Figure 1: Rank mapping in PREFER

EXAMPLE 1. *Fig. 1 shows 8 tuples: $t_1, t_2, \ldots, t_8$ in a tiny database. Each tuple has two attribute x and y. Suppose the pre-computed ranking order is built by the ranking function $x + y$. The order of each tuple in the index is determined by its projection onto the line $y = x$, which is orthogonal to $x + y = 0$. Similarly, a query with ranking function $3x + y$ corresponds to the line $y = 3x$. Suppose the query asks for top-2, and the results are t_2 and t_1. However, t_1 is ranked last with respect to $x+y$. That is, the system has to retrieve all tuples in the database to answer the query.*

The layered indexing methods are less sensitive to the query weights. Generally, they organize data tuples into consecutive layers according to the geometry layout, such that any top-k queries can be answered by up to k layers of tuples. Thus the worst case performance of any top-k query can be bounded by the number of tuples in the top k layers. The representative work in this category is the Onion technique [5], which greedily computes convex hulls on the data points, from outside to inside. Each tuple belongs to one layer. The query processing algorithm is able to leverage the domination relationships between consecutive layers and may stop earlier than the k^{th} layer is touched. For example, if the best rank of tuples in the c^{th} layer is no less than k among all tuples in the top-c layers ($c \leq k$), then all the tuples behind the c^{th} layer need not to be touched because they cannot rank before k. However, in order to exploit this domination relation between the consecutive layers, each layer is constructed conservatively and some tuples are unnecessarily to be put in top layers (as demonstrated in example 2) [2].

EXAMPLE 2. *Using the same sample database in Example 1, the constructed convex shells are shown in Fig. 2 (a). There are two layers constructed on the 8 tuples and for any top-2 query, all 8 tuples will be retrieved. In fact, we can exploit more layer opportunities in this example. Fig. 2 (b) shows an alternative layer construction which has four layers: $\{t_1, t_2, t_3, t_4, t_5\}$, $\{t_7\}$,$\{t_8\}$ and $\{t_6\}$ (solid lines). Tuple t_6 can be put in the 4^{th} layer because for any linear query with non-negative weights, t_3 must rank before t_6, one of the tuples in $\{t_2, t_4\}$ must rank before t_6 and one of the tuples in $\{t_1, t_7\}$ must rank before t_6 (dashed lines). These claims can be verified by linear algebra and we will discuss the details later in this paper. For the same reason, t_8 can be put in 3^{rd} layer. Any top-2 query on this layered index will only retrieve 6 tuples.*

[2]Since we assume all query weights are non-negative, we can improve the onion technique by constructing convex shells, instead of convex hulls for each layer. A convex shell is a partial convex hull where only those surfaces which can be seen by the origin $(0, 0, \ldots, 0)$ are kept (assume all tuples have non-negative values). Those tuples which are in the unseen surfaces will participate to build the next layers.

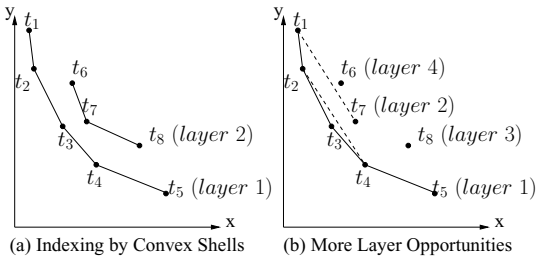

Figure 2: Multi-Layered Index

A key observation is that it may be beneficial to push a tuple as *deeply* as possible so that it has less chance to be touched in query execution. Motivated by this, we propose a new criterion for sequentially layered indexing: *for any k, the number of tuples in top k layers is minimal in comparison with all the other layered alternatives.* Since any top-k query can be answered by at most k layers, our proposal aims at minimizing the worst case performance on any top-k queries. Hence the proposed index is *robust*. Table 1 shows the minimal, maximal and average number of tuples retrieved from the databases to answer top-50 queries on a real data set and a synthetic data set using PREFER, Onion (with convex shell) and our proposed method (robust index)[3]. While both PREFER and Onion are sensitive to the query weights, our method, though not optimal in some cases, has the best expected performance.

Index	Real Data			Synthetic Data		
Methods	Min	Max	Avg	Min	Max	Avg
PREFER	89	2133	609.5	99	2468	1434.8
Onion	542	728	660.6	524	724	626.3
Robust	375	375	375	510	510	510

Table 1: Number of tuples retrieved to answer top-50 queries

Another appealing advantage of our proposal is that the top-k query processing can be seamlessly integrated into current commercial databases. Both Onion and PREFER methods require the advanced query execution algorithms, which are not supported by many database query engines so far. Our proposal transfers most computational efforts into the index building step. As soon as the tuples are sorted by the

[3]The real data set is a fragment of Forest Covertype data [3] with 3 selected dimensions: Elevation, HDTR and HDTFP. The synthetics data has 3 dimension with uniform distribution. Both data has $10k$ tuples. We issue 10 queries by randomly choosing the weights w_1, w_2, w_3 from $\{1, 2, 3, 4\}$.

computed layers in the database, any top-k query can be answered by simply issuing an SQL-like statement:

$$\begin{aligned} &select\ top\ k\ *\ from\ D \\ &where\ layer <= k \\ &order\ by\ f_{rank} \end{aligned}$$

The main contributions of this paper are summarized as follows.

1. We derive a necessary and sufficient condition for *robust index*: A tuple t can be put in the *deepest* layer l if and only if (a) for any possible linear queries, d is not in top $l-1$ results; and (b) there exists one query such that d belongs to top l results.
2. We show that there is an $O(n^d \log n)$ algorithm to compute the deepest layer for all tuples, where n is the size of the database and d is number of dimensions.
3. To reduce the high complexity of the exact solution, we propose an approximate method to compute the robust index. The approximate approach has time complexity $O(2^d n(\log n)^{r(d)-1})$, where $r(d) = \lceil \frac{d}{2} \rceil + \lfloor \frac{d}{2} \rfloor \lceil \frac{d}{2} \rceil$.

The rest of the paper is organized as follows. We first discuss the related work in Section 2. Section 3 gives the problem statement. The optimal solution is presented in Section 4 and the approximate alternative is described in Section 5. Our experimental results are shown in Section 6. We discuss the possible extensions in Section 7 and conclude our work in Section 8.

2. RELATED WORK

Previous work on rank-aware indexing can be classified into three categories: the distributive indexing (*i.e.* sort-merge) [8, 9, 10], the spatial indexing [12], and the sequential indexing [5, 13]. Our work falls in the class of sequential indexing. Here we briefly discuss the other two approaches. The distributive indexing approach sorts individual attributes separately, and during query execution, attributes from different lists are merged and evaluated by the ranking function. The algorithm assumes that the query function is monotone. A threshold algorithm is developed to determine the early stop condition. One important distinction between this approach and other indexing methods is that distributive indexing does not exploit attribute correlation. This penalizes query performance. The spatial indexing approach applies spatial data structure such as R-tree or k-d-B tree. Data points are stored in R-tree. At query time, the algorithm does not seek top-k results directly, instead, the query is processed by retrieving all the tuples that are greater than some threshold. Retrieved tuples are evaluated and sorted for final top-k results. The main difficulty of this approach lies in determining the threshold to prune the search space. A loose (tight) threshold may lead too much (few) returns. It also does not have the progressive property. Adjusting to new threshold causes the whole procedure to start from scratch.

The layered indexing methods for linear constraint queries have been studied in the computational geometry community. An example work is [2], where the authors proposed to minimize the number of disk blocks used to store the points and the number of disk access needed to answer a half-space range query. The major difference between this paper and their work is that we focus on top-k query processing in relational database and the data is accessed sequentially, starting from the first layer. While in [2], one tuple can appear in multiple layers and the query processing algorithm may access the same tuple multiple times.

Our proposal aims at exploiting domination relations between tuples. A closely related work is the skyline tuples [4, 7], where the one (tuple) to one (tuple) domination is studied. In this paper, we generalize the one to one domination to many (tuples) to one (tuple) domination, and thus more domination relations can be exploited. Moreover, research work in skyline tuples only computes one layer of tuples, while here we propose an efficient method to compute multiple layers for the entire database.

3. PROBLEM STATEMENT

We first define the notations in the context of linear ranked query. The task of finding top-k tuples from a database can be posed with either maximization or minimization criterion. Since a maximal query can be turned into a minimal one by switching the sign of objective function, in the rest of the paper we assume *minimization queries* are issued. Let $R = (t_1, t_2, \ldots, t_n)$ be a relation with d attributes $(A_1, A_2, \ldots, A_d)$ whose domains are real values, and let t^i refer to the value of attribute A_i in the tuple $t \in R$. For simplicity, we assume that there are no duplicate values on any attribute (Ties can be easily broken by comparing a unique *tid* assigned to each tuple).

A *ranked query* q consists of an evaluation function f where $f(t)$ defines a numeric score for each tuple $t \in R$. The output of query q is the ranked sequence $[t'_1, t'_2, \ldots, t'_n]$ of tuples in R ($|R| = n$) such that $f(t'_1) \leq f(t'_2) \leq \ldots \leq f(t'_n)$. In this paper, we focus on queries using *linear combination function* $f(t) = \sum_{i=1}^{d} w_i t_i$, where $w = (w_1, w_2, \ldots, w_d)$ is a weighting vector. We further assume that the evaluation function f is *monotone* [10] (*i.e.* $f(a) = f(a^1, a^2, \ldots, a^d) \leq f(b^1, b^2, \ldots, b^d) = f(b)$ whenever $a^i \leq b^i$ for every $i \in \{1, 2, \ldots, d\}$). We will discuss the extension to non-monotone queries in section 7. Without loss of generality, let each value in the weighting vector be normalized in $[0, 1]$ and $\sum_{i=1}^{d} w_i = 1$. A *top-k query* is a ranked query which only asks for top-k ranked results $[t'_1, t'_2, \ldots, t'_k]$.

The *sequentially layered index* and *robust index* can be defined as follows.

DEFINITION 1. *(**Sequentially Layered Index**) A sequentially layered index L of a relation R partitions all tuples in R into consecutive multiple layers $L(R) = [l_1, l_2, \ldots, l_m]$ such that: (1) $l_i \cap l_j = \phi$ $(\forall i \neq j)$ and $\bigcup_{i=1}^{m} l_i = R$; and (2) any linear top-k query can be answered by $L_k = \bigcup_{i=1}^{k} l_i$.*

DEFINITION 2. *(**Robust Index**) A sequentially layered index $L^*(R) = [l^*_1, l^*_2, \ldots, l^*_{m^*}]$ of a relation R is robust if for any other sequentially layered index $L(R) = [l_1, l_2, \ldots, l_m]$, $L^*_k \subseteq L_k$, for any $k > 0$.*

Given a sequentially layered index L, let the layer which the tuple $t \in R$ belongs to be $l(t, L)$. In the rest of the paper, we refer L^* the robust index and note $l(t, L^*)$ as the *robust layer* of t. The next theorem gives a necessary and sufficient condition for robust index.

THEOREM 1. *A sequentially layered index L is robust if and only if for each tuple $t \in R$, $l(t, L)$ satisfies: (1) for any possible linear queries, t does not rank in top $l(t, L) - 1$; and (2) there exists at least one linear query such that t ranks top $l(t, L)$.*

Sketch of Proof. Let us call L the layered index which satisfies the above two conditions and L' an arbitrary sequentially layered index. We show that $L_k \subseteq L'_k$ $(\forall\ k > 0)$. For each $t \in R$, since there is at least one query q such that t belongs to top $l(t, L)$, we have $l(t, L') \leq l(t, L)$. Otherwise, L' is not able to give top-$l(t, L)$ results for query q. We conclude $L_k \subseteq L'_k$ for any $k > 0$ and thus L is robust. ∎

The two conditions stated in Theorem 1 are equivalent to that $l(t, L^*)$ is the minimal ranking of t over all possible queries. Our problem of *robust indexing* is defined as below.

DEFINITION 3. ***(Robust Indexing)*** *Given a relation R, compute the robust index L^* for all $t \in R$ such that $l(t, L^*)$ is the minimal ranking of t over all linear queries.*

4. EXACT SOLUTION

This section discusses the exact solution for the *robust indexing* problem. For any $t \in R$, our goal is to find the minimal ranking of t over all linear queries (*i.e.*, robust layer of t).

Consider the case where the number of dimension $d = 1$. We can sort tuples in R completely and each tuple occupies a layer. This takes $O(n \log n)$. Suppose $d = 2$, for an arbitrary tuple $t \in R$, as shown in Figure 3, A_1 and A_2 are two attributes, and $t_j (j = 1, 2, \ldots, 6)$ are all other tuples in R. We need to compute the minimal ranking of t over all possible linear queries q. Suppose the query weighting vector is $w = (w_1, w_2)$ ($w_1, w_2 \in [0, 1]$ and $w_1 + w_2 = 1$) and W is the set of all valid assignments of w. A naïve way is to enumerate all possible assignment of $w \in W$ and compute the minimum ranking for t. This is not possible because the number of possible linear queries (*i.e.*, $|W|$) is infinite.

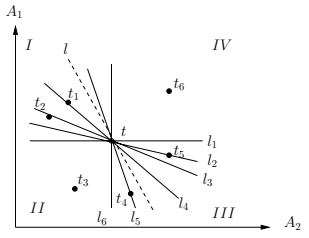

Figure 3: Exact Solution for $d = 2$

On the other hand, one can create the one-to-one correspondence between any weighting vector w and a line l which crosses the tuple t. The ranking of t with respect to w is determined by the number of tuples at the left-bottom side of the line l. For example, in Figure 3, the dashed line l corresponds to a weighting vector w and the ranking of t over w is 5 (since there are four tuples t_1, t_2, t_3 and t_4 at the left-bottom side). Moreover, the constraints of $w_1, w_2 \in [0, 1]$ further restrict that the line must cross the regions I and III (as shown in Figure 3).

With the one-to-one correspondence, we can partition W into finite intervals such that in each interval W_i, all the weighting vectors $w \in W_i$ generate the same ranking results for t. Using the example in Figure 3, we can partition W into 5 intervals and the boundaries are l_1, l_2, l_3, l_4, l_5 and l_6. More specifically, the boundary lines are computed by the horizontal line l_1, vertical line l_6 and lines linking t and other tuples in the subregions I and III. Clearly, the weighting vectors within each interval generate the same ranking results for t. To compute the minimum ranking of t, we only need to check the ranking results on those boundaries. One can sort the boundary lines by their angles to l_1, and then traverse the lines in the order to obtain the exact value of the minimum ranking of t. This step takes time $O(n \log n)$. We conclude that computing minimum rankings for all tuples in R takes $O(n^2 \log n)$ time.

Generally, for $d > 2$, we have the following theorem.

THEOREM 2. *Given a relation R, the robust indexing problem can be solved within $O(n^d \log n)$ time complexity, where $n = |R|$ is the number of tuples, and d is the number of dimensions.*

Proof. See Appendix. ∎

The $O(n^d \log n)$ complexity makes the exact solution unattractive in real applications. In the next section, we discuss an alternative approach which approximates the minimum ranking of each tuple in R. To ensure that any top-k queries can be answered by the top k layers of tuples without false positives, the approximate layer $l(t, \hat{L})$ for any tuple t should satisfy $l(t, \hat{L}) \leq l(t, L^*)$, where $\hat{L}, L^*$ are the approximate and exact robust index, respectively.

5. APPROXIMATE SOLUTION

This section presents the method to compute the approximate (*i.e.*, lower bound) robust layer for each tuple $t \in R$. Given a tuple t, the exact solution will first sort other tuples and find the interval boundaries. This step is quite expensive, especially when the number of dimensions is large. Instead of computing those exact boundaries, the approximate algorithm creates the boundaries by evenly partitioning the interesting regions (*i.e.*, region I and III in Figure 4).

Using the 2-dimensional example in Figure 4, we outline the major steps of the approximate algorithm as follows:

Partition the region I and III evenly into B sub-regions (*e.g.*, $I_1, I_2, \ldots, I_B$ and $III_1, III_2, \ldots, III_B$).
Count the number of tuples in region II and sub-regions $I_1, I_2, \ldots, I_B$ and $III_1, III_2, \ldots, III_B$.
Match the number of tuples in sub-regions and compute the lower bound of the robust layer for each tuple.

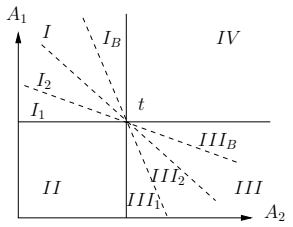

Figure 4: Approximate By Partitioning

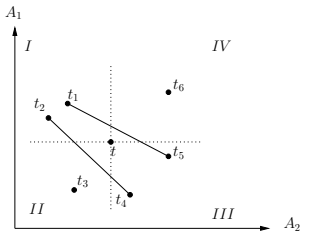

Figure 5: Domination Set

Particularly, the *partitioning* and *matching* steps are associated with each other (*e.g.*, the match strategy determines how the regions are partitioned). In the rest of this section, we first present our *partitioning* and *matching* strategy, followed by an efficient algorithm for *counting* step. Finally, we present the complete algorithm.

5.1 Partitioning and Matching

We first introduce a new concept: *domination set*, and then show that the robust layer of a tuple t can be lower bounded by the number of exclusive domination sets of t. The main ideas are demonstrated with $d = 2$ and the generalization to $d \geq 3$ is discussed in the end of this subsection.

5.1.1 Domination Set

We first define the *domination* and *domination set*.

DEFINITION 4. *(**Domination**) A tuple $t \in R$ dominates another tuple $s \in R$ if $t^i \leq s^i$ for all $1 \leq i \leq d$, where d is the number of dimensions of R.*

If a tuple t dominates another tuple s, then for any monotone linear query q with evaluation function f, we have $f(t) \leq f(s)$.

DEFINITION 5. *(**Domination Set**) A set of tuples $DS = \{t_1, t_2, \ldots, t_p\}$ is a domination set of tuple t if there exists linear weights $\{v_1, v_2, \ldots, v_p\}$, where $v_i \in [0, 1]$ and $\sum_{i=1}^{p} v_i = 1$, such that $t' = \sum_{i=1}^{p} v_i t_i$ dominates t. A domination set DS is* minimal *if any subset of DS is NOT a dominating set of t.*

A domination set $DS = \{t_1, t_2, \ldots, t_p\}$ is also noted as p-domination set. Given a relation R with d dimensions, one can derive the following conclusion from the theorem of linear independence: if a p-dominating set is *minimal*, then $p \leq d$. In the rest of the paper, we assume all referred domination sets are minimal. We say two domination sets DS_i and DS_j are *exclusive* if $DS_i \cap DS_j = \phi$. An example of domination set is shown as follows.

EXAMPLE 3. *In Fig. 5, suppose t is the tuple under study. Tuple t_3 dominates t since on both dimensions A_1 and A_2. The values of t_3 are less than those of t. Tuples t_2 and t_4 constitute a 2-domination set of t. Note that the valid linear combinations of t_2 and t_4 (as defined in the domination set) are the segment linking t_2 and t_4. A pair of tuples constitute a 2-domination set of t if t is at the right-upper side of the segment. Correspondingly, $\{t_1, t_5\}$ is not a domination set of t. Moreover, $\{t_3, t_4\}$ is not a minimal domination set since t_3 only can dominate t. One can further verify that t_6 (as well as all other tuples in region IV) is not in any domination set of t.*

The following lemma demonstrates a property of domination set.

LEMMA 1. *Let $DS = \{t_1, t_2, \ldots, t_p\}$ be a p-domination set of tuple t, then $l(t, L^*) > \min_{i=1}^{p} l(t_i, L^*)$, where L^* is the robust index.*

Proof. It is equivalent to show that for any linear query q with evaluation function f, there exists a tuple $t' \in DS$ such that $f(t') \leq f(t)$.

Assume there is a query q such that for all $t' \in DS$, $f(t') > f(t)$. Then for any linear weights $\{v_1, v_2, \ldots, v_p\}$, such that $v_i \in [0, 1]$ and $\sum_{i=1}^{p} v_i = 1$, we have $\sum_{i=1}^{p} v_i f(t') > f(t)$. On the other hand, since f is linear, we have $\sum_{i=1}^{p} v_i f(t') = f(\sum_{i=1}^{p} v_i t')$. According to the definition of domination set, there exists a linear weight $\{u_1, u_2, \ldots, u_p\}$ such that $f(\sum_{i=1}^{p} u_i t') \leq f(t)$. We thus have $\sum_{i=1}^{p} u_i f(t') \leq f(t)$. This contradicts with the assumption. ∎

Suppose $d = 2$, for any tuple $t \in R$, every other tuple in region II (*i.e.*, the left-bottom corner in Figure 5) consists of a 1-domination set. Tuples in region IV are not in any domination sets. Tuples in region I and region III can be paired to constitute 2-domination sets. Generally, let $\mathcal{DS}^1(t)$ be the set of all 1-domination sets of t, and $\mathcal{DS}^2(t)$ be the set of all 2-domination sets of t. Assume $\mathcal{EDS}^2(t) \subseteq \mathcal{DS}^2(t)$ is the largest subset of $\mathcal{DS}^2(t)$ such that all 2-domination sets in $\mathcal{EDS}^2(t)$ are mutually exclusive. The following lemma shows that the robust layer of a tuple t can be lower bounded by the number of domination sets.

LEMMA 2. *Given a relation R, for any tuple $t \in R$, the robust layer $l(t, L^*) > |\mathcal{DS}^1(t)| + |\mathcal{EDS}^2(t)|$, where L^* is the robust index.*

Proof. Given any linear query q with evaluation function f, by the definition of domination set, we know any tuple in the 1-dominating set ranks before t. According to Lemma 1, at least one tuple in a 2-dominating set ranks before t. Since all tuples in $\mathcal{DS}^1(t)$ do not appear in $\mathcal{EDS}^2(t)$ (otherwise, the corresponding 2-domination set is not minimal) and tuples

in $\mathcal{EDS}^2(t)$ are mutually exclusive, we conclude that there are at least $|\mathcal{DS}^1(t)|+|\mathcal{EDS}^2(t)|$ tuples ranked before t, thus $l(t, L^*) > |\mathcal{DS}^1(t)| + |\mathcal{EDS}^2(t)|$. ∎

Involving p-domination sets ($p \geq 3$) gives better approximation, but increases computational complexity as well. In this paper, we use up to 2-domination set to lower bound $l(t, L^*)$. Using Lemma 2, we can lower bound the value of $l(t, L^*)$ by $|\mathcal{DS}^1(t)|$ and $|\mathcal{EDS}^2(t)|$. $|\mathcal{DS}^1(t)|$ is simply the number of tuples in region I. However, computing $|\mathcal{EDS}^2(t)|$ is not an easy task. Generally, the problem of finding $\mathcal{EDS}^2(t)$ is similar to the *maximal matching* problem [15] in a bipartite graph where the computational complexity is $O(n^3)$. Instead of computing the exact value of $|\mathcal{EDS}^2(t)|$, we present a simple matching method to compute the lower bound the value of $|\mathcal{EDS}^2(t)|$.

5.1.2 Matching

We first demonstrate our method with $d = 2$, and then generalize it to $d \geq 3$.

Consider the case where $d = 2$, for each tuple t, we use $B-1$ lines to evenly partition regions I and III into B subregions (as shown in Figure 4) such that every tuple in I_i (III_i) can be paired with any tuple in $III_1, III_2, \ldots, III_{B-i}$ ($I_1, I_2, \ldots, I_{B-i}$) to form a 2-dominating set (since the segments between the paired tuples lie at the left-bottom side of t). We have the following lemma.

LEMMA 3. *Given a relation R with $d = 2$, for each tuple t, $|\mathcal{EDS}^2(t)| \geq \min(\sum_{i=1}^{B-1}|I_i|, |III_1|+\sum_{i=1}^{B-2}|I_i|, \sum_{i=1}^{2}|III_i| + \sum_{i=1}^{B-3}|I_i|, \ldots, \sum_{i=1}^{B-1}|III_i|)$.*

Proof. We show the case $\sum_{i=1}^{B-1}|I_i| = \min(\sum_{i=1}^{B-1}|I_i|, |III_1|+\sum_{i=1}^{B-2}|I_i|, \sum_{i=1}^{2}|III_i| + \sum_{i=1}^{B-3}|I_i|, \ldots, \sum_{i=1}^{B-1}|I_i|)$, the proof for other cases are similar. From $\sum_{i=1}^{B-1}|I_i| \leq |III_1|+\sum_{i=1}^{B-2}|I_i|$, we have $|I_{B-1}| \leq |III_1|$. Using the same argument in Lemma 2, all tuples in $|I_{B-1}|$ can find a different tuple in $|III_1|$ to form mutually exclusive 2-domination sets. Since $|I_{B-1}| \leq |III_1|$, there are tuples left in III_1 after pairing with those in I_{B-1}. Let the set of rest tuples be III_1'. Similarly, from $\sum_{i=1}^{B-1}|I_i| \leq \sum_{i=1}^{2}|III_i| + \sum_{i=1}^{B-3}|I_i|$, we have

$$|I_{B-2}| + |I_{B-1}| \leq |III_1| + |III_2|$$
$$\Rightarrow |I_{B-2}| \leq |III_1| + |III_2| - |I_{B-1}| = |III_2| + |III_1'|.$$

We conclude that all tuples in I_{B-2} can find a different tuple in $III_1' \cup III_2$ to form mutually exclusive 2-domination sets. Continuing with this procedure, for each tuple $t \in \bigcup_{i=1}^{B-1} I_i$, we can find a different tuple in region III to form a mutually exclusive 2-domination set. Hence, $|\mathcal{EDS}^2(t)| \geq \sum_{i=1}^{B-1}|I_i|$. ∎

Based on Lemmas 2 and 3, we can lower bound the value of $l(t, L^*)$ by aggregating $|\mathcal{DS}^1(t)|$ and the lower bound value of $|\mathcal{EDS}^2(t)|$. Suppose the approximate method construct an index $\hat{L}$ and the layer of t is $l(t, \hat{L})$. The following theorem states the quality of the approximation method.

THEOREM 3. *Given a relation R with $d = 2$, suppose the data forms a uniform distribution and regions I and III are partitioned evenly into B subregions. For each tuple t, we have*

$$E[\frac{l(t, \hat{L})}{l(t, L^*)}] \geq 1 - \frac{1}{B}$$

where $E[\frac{l(t,\hat{L})}{l(t,L^*)}]$ is the expected approximation quality.

Proof. See Appendix. ∎

5.1.3 Three or Higher Dimensions

We now discuss how to extend Lemma 3 to cases where $d \geq 3$. Given a tuple $t = (t^1, t^2, \ldots, t^d)$ with $d \geq 3$, we have 2^d subspaces characterized by their relationship to t. The first subspace is $000\ldots0$ (d bits), such that for each tuple t' in this subspace, $t'^i \leq t^i$ ($i = 1, 2, \ldots, d$). for any other subspaces a, if the i^{th} bit is set as 1, then for each tuple $t' \in a$, $t'^i > t^i$. Generally, a 0-bit corresponds to a *dominating dimension* and a 1-bit corresponds to a *dominated dimension*.

Subspace 0 contains all the tuples dominating t (thus forms the 1-domination sets), while subspace 2^d-1 contains all the tuples dominated by t (thus has no use for computing t's robust layer). We group all the other subspaces into $2^{d-1}-1$ pairs, and the i^{th} subspace is paired with $(2^d-1-i)^{th}$ subspace. For each paired subspace (a, b), the set of dominating dimensions of a is identical to the set of dominated dimensions of b, and vice versa. Let the set of dominating dimensions of a be $\{i_1, i_2, \ldots, i_l\}$, and the set of dominated dimensions of a is $\{j_1, j_2, \ldots, j_g\}$ ($l+g = d$). In order to get a similar lower bounding method as stated in Lemma 3, we first create Eqn. (1) and Eqn. (2) to construct partitions in subspaces a and b.

a_p-partition:

$$\begin{cases} t'^j \geq t^j & j \in \{j_1, \ldots, j_g\} \\ \gamma_p t'^i + t'^j \leq \gamma_p t^i + t^j & i \in \{i_1, \ldots, i_l\}, j \in \{j_1, \ldots, j_g\} \end{cases} \quad (1)$$

b_p-partition:

$$\begin{cases} t'^i \geq t^i & i \in \{i_1, \ldots, i_l\} \\ \gamma_p t'^i + t'^j \leq \gamma_p t^i + t^j & i \in \{i_1, \ldots, i_l\}, j \in \{j_1, \ldots, j_g\} \end{cases} \quad (2)$$

where γ_p ($p = 1, 2, \ldots, B-1$) satisfies $\gamma_1 < \gamma_2 < \ldots < \gamma_p$. The partitioning equations can be understood as follows: (1) the $t'^j \geq t^j$ or $t'^i \geq t^i$ equations are simply the boundary constraints of subspaces a and b; and (2) the $\gamma_p t'^i + t'^j \leq \gamma_p t^i + t^j$ equations are a set of hyperplanes which evenly partition the subspaces. One can futher verify that $a_1 \subseteq a_2 \subseteq \ldots a_{B-1} \subseteq a$ and $b_{B-1} \subseteq b_{B-2} \subseteq \ldots b_1 \subseteq b$.

Let $I_i = a_i - a_{i-1}(a_B = a, a_0 = \phi)$, and $III_i = b_{B-i} - b_{B+1-i}(b_B = \phi, b_0 = b)$, $i = 1, 2, \ldots, B$. We have the following lemma.

LEMMA 4. *Given a pair of d-dimensional subspaces (a, b) w.r.t. $t \in R$, I_i (III_i) ($i = 1, 2, \ldots, B$) partition a (b) into B un-overlapping subregions such that every tuple in I_i (III_i) can be paired with any tuple in $III_1, III_2, \ldots, III_{B-i}$ ($I_1, I_2, \ldots, I_{B-i}$) to form a 2-domination set of t.*

Proof. See Appendix. ∎

Using Lemma 4, one can apply the same matching method in Lemma 3 on three or higher dimensions. Eqn. (1) consists of $g+lg$ inequalities and Eqn. (2) consists of $l+lg$ inequalities. Since $l+g=d$, by simple calculation, one can verify that both of them are upper bounded by $\lceil \frac{d}{2} \rceil + \lfloor \frac{d}{2} \rfloor \lceil \frac{d}{2} \rceil$ (referred as $r(d)$).

5.2 Counting

To compute the approximate result, we need to know the number of tuples in region II (for the value of $|\mathcal{DS}^1(t)|$) and the number of tuples in subregions I_i and III_i (for approximating the value of $|\mathcal{EDS}^2(t)|$). The first observation is that both problems share the same property and can be solved by a single algorithm. We first give a formal definition for the counting problem.

DEFINITION 6. *(**Domination Factor**) Given a relation R with d dimensions, for each tuple $t \in R$, the domination factor of t is $DF(t) = |S|$, where S is the set of tuples which dominate t.*

Note although domination factor directly corresponds to 1-domination set, it can also be used to compute the lower bound of $|\mathcal{EDS}^2(t)|$ where the values of $|I_i|$ ($|III_i|$) can be seen as values of $DF(t)$ in the linearly transformed spaces, as demonstrated in the following example.

EXAMPLE 4. *In Figure 4, the sub-region I_1 can be described as $\{t'|t'^1 \geq t^1$ and $w_1t'^1 + w_2t'^2 \leq w_1t^1 + w_2t^2\}$, where (w_1, w_2) corresponds to the boundary line between I_1 and I_2. After transforming the original space (A_1, A_2) to $(A'_1 = -A_1, A'_2 = w_1A_1 + w_2A_2)$, the value of $|I_1|$ is exactly the value of $DF(t)$ in the transformed space.*

The naïve solution for the domination factor problem is for each tuple t, to scan the database and count the number of tuples dominating t. This takes $O(n^2)$. Here we present an improved algorithm. The input of domination factor problem is the transformed space where the number of dimensions is up bounded by $r(d)$ (see section 5.1.3). For simplicity, we still use d to refer to the number of dimensions in the transformed space. We first discuss a warm-up algorithm for $d = 2$, then present a *divide and conquer* approach for $d \geq 3$.

5.2.1 Two Dimension Case

Consider the case where $d = 2$, the conditions for t' dominating t are $t'^1 \leq t^1, t'^2 \leq t^2$. We sort all tuples in R with respect to the values in attribute A_1 (ascending order), and then progressively retrieve tuples t from R and maintain the values in attribute A_2 (*i.e.*, t^2) using a binary tree T. More specifically, whenever we get a new t, before we insert t^2 into T, we first query t^2 in T to find the number of tuples whose A_2 value is no larger than t^2. Since tuples are sorted by A_1 values, this number is exactly the domination factor of t.

The algorithm needs a binary tree which can return the number of records whose values are no larger than a query value in $O(\log n)$ time. To achieve this, we modify the traditional AVL-tree [11] by adding a new field $Left$ to each node N. The $Left$ value indicates the number of records (including N) in N's left subtree. The modifications on insertion and rotation with respect to $Left$ are straightforward [11]. At query time, when a binary traversal reaches a node N whose value is no larger than the query value, we can accumulate $N.Left$ to the final answer without going to the left sub-tree of N. The complexities of insertion and query on the modified AVL-tree are kept same as $O(\log n)$. The algorithm is described as in Algorithm 1. The complexity of the algorithm is $O(n \log n)$.

Algorithm 1 Domination Factor: $d = 2$

Input: A Relation R with $d = 2$
Output: For each $t \in R$, compute $DF(t)$

1: Sort R on attribute A_1 (value ascending order);
2: Initialize a modified AVL-tree, T;
3: **for** each $t \in R$ (retrieved sequentially)
4: Query T, and let $DF(t)$ be the number of tuples whose values are no larger than t^2;
5: Insert t^2 into T;
6: **return**;

5.2.2 Divide and Conquer

Here we introduce a divide and conquer approach for $d \geq 3$. The algorithm is described in Algorithm 2. The two main procedures are *partition* and *merge*. The algorithm starts from the first attribute, and recursively partition the following attributes (in lines 16-18). In the partitioning step, the input tuples set P is divided into two subsets P_1 and P_2 according to attribute A_s. In the merging step, the algorithm updates the domination factor of tuples in P_2 by merging P_1.

We discuss three different cases in the merging step: (1) P_1 (or P_2) contains only one tuple: We can simply do a linear scan over P_2 (or P_1) and update domination factors for tuples in P_2. (2) $s = d - 1$: At that time, there are only two attributes A_{d-1} and A_d on which the relations between $t_1 \in P_1$ and $t_2 \in P_2$ have not been exploited (on attributes $A_i, i < s$, we already have $t_1^i \leq t_2^i$). In this case, we can use a similar approach to Algorithm 1. We first merge tuples from P_1 and P_2, and sort them by the attribute value A_{d-1}; then use a modified AVL-tree T to maintain values of A_d. The difference from algorithm 1 is that we only want to compute domination factors of P_2 tuples from P_1. For this purpose, in the merged tuple list, when we get a P_1 tuple, we only insert it into T without query; and when we get a P_2 tuple, we only query on T without inserting. The complexity of the whole procedure is $O((|P_1| + |P_2|) \log(|P_1| + |P_2|))$. And (3) otherwise, we partition P_1 and P_2 using the median tuple $t_m \in P_2$ (note P_2 is pre-sorted by dimension s). P_{21} and P_{22} are two sub-partitions (of P_2) divided by t_m. All tuples (in P_1) whose value on dimension s is no larger than t_m go to P_{11}, and the rest form P_{12}. Since for any $t_{12} \in P_{12}$ and $t_{21} \in P_{21}$, we have $t_{12}^s > t_{21}^s$, thus P_{12} has no domination effect on P_{21}. Hence, we only need to recursively merge $(P_{11}, P_{21}), (P_{12}, P_{22})$, and (P_{11}, P_{22}). Furthermore, for (P_{11}, P_{22}), since for all tuples $t_1 \in P_{11}$ and $t_2 \in P_{22}$, we have $t_1^s \leq t_2^s$ on dimension s, we can skip dimension s and go to next dimension $s + 1$ for the next merging step.

Algorithm 2 Domination Factor: divide and conquer

Input: Relation R with $d \geq 3$
Output: For each $t \in R$, compute $DF(t)$

1: Sort R on attribute A_1 (value ascending order);
2: Call $Partition(1, R)$;
3: **return**;

Procedure **Partition(s,P)** //P is sorted by A_s
4: **if** ($|P| == 1$) **return**;
5: $P_1 = \{t_1, t_2, \ldots, t_{|P|/2}\}$;
6: $P_2 = \{t_{|P|/2+1}, t_{|P|/2+2}, \ldots, t_{|P||}\}$;
7: Call $Partition(s, P_1)$ and $Partition(s, P_2)$;
8: Sort P_2 on attribute A_{s+1};
9: Call $Merge(s+1, P_1, P_2)$;
10: **return**;

Procedure **Merge(s,P_1,P_2)** // P_2 is sorted by A_s
11: **else if** ($|P_1| == 1 || |P_2| == 1$)
12: Linear scan P_1 or P_2;
13: **else if** ($s == d-1$)
14: Binary search P_1, P_2;
15: **else**
16: $P_{21} = \{t_1, \cdots, t_{|P_2|/2}\}, P_{22} = \{t_{|P_2|/2+1}, \cdots, t_{|P_2|}\}$;
17: $P_{11} = \{t | t^s \leq t^s_{|P_2|/2}, t \in P_1\}$;
18: $P_{12} = \{t | t^s > t^s_{|P_2|/2}, t \in P_1\}$;
19: Call $Merge(s, P_{11}, P_{21})$ and $Merge(s, P_{12}, P_{22})$;
20: Sort P_{22} on Attribute A_{s+1};
21: Call $Merge(d+1, P_{11}, P_{22})$;
22: **return**;

This is a typical divide and conquer algorithm and the complexity analysis can be found in many previous work (*i.e.*, [14]). We state the following theorem without proof.

THEOREM 4. *For $d \geq 3$, the complexity of the algorithm 2 is $O(n(\log n)^{d-1})$.*

5.3 The Complete Algorithm

We present the complete algorithm as a summary of the approximate approach. The algorithm assumes the data retrieved from disk fits in main memory.

The algorithm first computes the $|\mathcal{DS}^1(t)|$ value for each $t \in R$ by calling the couting procedure: *DF*; and then approximates the $|\mathcal{EDS}^2(t)|$ value by looking at the $2^{d-1}-1$ subspace pairs. Each subspace is partitioned into B sub-regions and the number of tuples in every sub-region is also computed by the *DF* procedure. Lemma 3 is used to lower bound the value of $|\mathcal{EDS}^2_p(t)|$ for subspace pair p. The main computational step is the *DF* procedure, and it is called $O(2^d B)$ times. The overall complexity is $O(2^d Bn(\log n)^{r(d)-1})$, where $r(d) = \lceil \frac{d}{2} \rceil + \lfloor \frac{d}{2} \rfloor \lceil \frac{d}{2} \rceil$.

6. EXPERIMENTAL RESULTS

Here we report the experimental results of the approximate solution for robust index (referred as *AppRI*). We compare the performance with the Onion [5] and *PREFER* [13] approaches. In the following subsections, we first introduce the experiment settings, and then present the results with respect to the index building cost and the query performance.

Algorithm 3 The Approximate Algorithm

Input: Relation R, Number of Partitions B
Output: For each $t \in R$, the approximate layer $l(t, \hat{L})$.

1: Call $DF(R)$, let $l(t, \hat{L}) = |\mathcal{DS}^1(t)|$ ($\forall t \in R$);
2: **for** $2^{d-1}-1$ pair of subspace $p = (a, b)$
3: **for** $i = 1$ to B;
4: Transform R to R' using Eqn. (1) and Eqn. (2);
5: Call $DF(R')$, and compute $|I_i|$ and $|III_i|$;
6: Compute $|\mathcal{EDS}^2_p(t)|$ using Lemma 3;
7: $l(t, \hat{L}) = l(t, \hat{L}) + |\mathcal{EDS}^2_p(t)|$ ($\forall t \in R$);
8: **return**;

We notice that PREFER is originally proposed to use multi-views (or, multiple indices) to answer top-k queries. Our method can be easily adapted to exploit the benefit of using multiple ranked views. We will also address this issue later in this section.

6.1 Experimental Setting

Both our method and Onion are implemented using C++. The PREFER system is obtained from the authors. All experiments are carried out on an Intel Pentium-4 3.2GHz system with 1G of RAM running Windows Server 2003. We use both synthetic and real data sets for the experiments. The real data sets we consider are *abalone3D* and *Cover3D*. The abalone3D data [3] has $4,177$ tuples with 3 selected dimensions of length, weight, and shucked_weight. The cover3D is a fragment of Cover Forest Data [3], and it has $10,000$ tuples with 3 selected dimensions on Elevation, Horizontal_Distance_To_Roadways, and Horizontal_Distance_To_Fire_Points. We also generate a number of synthetic data sets for our experiments using a modified data generator provided by [4].

We compare the three methods according to two criteria: *the cost to build the index*; and *the number of tuples* retrieved in answering top-k queries. We expect that *AppRI* performs better in the comparison of running time, because both *Onion* and *PREFER* need to do additional computation to determine the stop condition, while *AppRI* only needs to retrieve tuples.

We assume queries are monotone. This assumption is also made by *PREFER*. The original proposal of Onion builds hulls on tuples, and thus it is able to answer all linear queries, including both monotone ones and non-monotone ones. To make a fair comparison, we compare with a variant version of onion: *convex shell*. That is, for each original convex hull, only those surfaces which can be seen by origin $(0, 0, \ldots, 0)$ are kept as a layer, and the other tuples on unseen surfaces will participate in the construction of shells in next layers. In this way, there are less tuples in each layer. The variant method makes a significant improvement on query performance over the original Onion approach. We refer it as *Shell* thereafter.

6.2 Cost of Building Index

We compare the index building costs of *Onion*, *Shell*, *PREFER* and *AppRI*. Since our method needs to specify the number of partitions B, we first study the sensitivity of our proposed approach with respect to B. We run a set of experiments on a synthetic data with uniform distribution. The data has 3 dimensions and $10k$ tuples. The value of B is varied from 2 to 20. Figure 6 shows the numbers of tuples in top-50 layers, and Figure 7 shows the corresponding contruction time. We observe that the curve of number of tuples w.r.t. to B is close to the function $1 - \frac{1}{B}$ (as discussed in Theorem 3). Generally, the number of tuples decreases when B increases, which indicates that less number of tuples will be retrieved using a larger B. When $B > 10$, the benefit by increasing B is limited. The construction time, as analyzed in time complexity, is linear with B. In the following experiemnts, we set $B = 10$.

We further compare the cost of index building by *AppRI*, *Onion*, and *Shell*. All of these three methods build layers on tuples. *PREFER* needs to compute a linear weight to build the ranked view (*i.e.*, index in this paper). The criterion is that the selected weight has generally good coverage over all the other weighs (see [13] for detail). We do not combine the results of *PREFER* here because the *PREFER* system is implemented with JAVA and the computation time depends on the system parameters. However, we observe that using the system default parameters, it takes more than $1,200$ seconds to pre-process a synthetic data set (with $50k$ tuples), where *AppRI* uses less than 400 seconds to build the index. We generate 5 data sets with increasing size (from $10k$ to $50k$). All the data sets have 3 dimensions. The computation time for *Onion*, *Shell* and *AppRI* is reported in Figure 8. We observe that *AppRI* is much more efficient comparing with *Onion* and *Shell*. The computation of convex hull is expensive and the *Onion* and *Shell* need to compute multiple hulls iteratively. The *Shell* uses more time since it generates more layers than those in the *Onion*.

6.3 Query Performance

The second set of experiments tests query performance. We compare *AppRI* with *Onion*, *Shell* and *PREFER*. For each experiment, we report the number of tuples retrieved from the indexed database to answer top-k queries. We vary the value of k and for each top-k value, we issue 10 linear queries by randomly choosing the weights w_1, w_2 and w_3 from $\{1, 2, 3, 4\}$, and report the average number of tuples over all queries.

The first experiment is run on a synthetic data with uniform distribution. The data has $10k$ tuples and 3 dimensions. The average number of tuples retrieved is shown in Figure 9. We observe that *AppRI* performs best among all the alternatives. As we explained earlier, *Shell* is much better than *Onion* since it takes advantage of monotone assumption. In the following experiments, we only show the results of *Shell*. *PREFER* is very sensitive to the query weighting. For example, in top-10 queries, the minimum number of tuples retrieved is 11, while the maximum number of tuples retrieved is $1,950$. *Shell* is less sensitive. For top-10 queries, the minimum number is 147 and the maximum number is 220. *AppRI* retrieves the same number of tuples (180) for all top-10 queries.

The above data set bears uniform distribution. As we discussed in Section 2, an important motivation for building sequential index is to exploit the data correlations. Our second experiment studies the query performance with respect to the data correlations. The correlation is controlled by a parameter c ($c = 0$ corresponds to uniform distribution and increasing c means more correlation are introduced in the data generation). All the data sets have $10k$ tuples and 3 dimensions. The average number of tuples to answer top-50 queries is shown in Figure 10. We observe that all methods perform better when correlation increases. *AppRI* gets more benefits because there are more domination relations in the data and tuples can be pushed to deeper levels. *PREFER* is better because on correlated data, it is less sensitive to the query weights. For example, in the data set where $c = 1$, the minimum number of tuples retrieved by *PREFER* is 51 and the maximum number of tuples retrieved is 356. The gain of *Shell* on the correlated data is quite limited because its conservative layer construction criterion.

Our last experiment with the synthetic data is to study the query performance with respect to the data size. We generate a group of data sets with different sizes (from $10k$ to $50k$). Each data has 3 dimensions and the correlation parameter is 0.5. The number of tuples retrieved for top-50 query is shown in Figure 11. It is interesting to see that the number of tuples retrieved by *Shell* is not monotonically increasing with the data size. This may be caused by the query algorithm used by *Shell*. With a larger data set, although the number of tuples in each layer increase, the *Shell* query algorithm may decide to stop at an earlier layer. The number of tuples retrieved by *AppRI* increases slightly with the data size.

Finally, we examine the query performances on the two real data sets: *abalone3D* and *Cover3D* (Section 6.1). The average number of tuples retrieved w.r.t. different top-k are shown in Figure 12 and Figure 13. We observe that in both real data sets, *AppRI* performs the best.

6.4 Multiple Views

In the final set of experiments, we explore the opportunities to use multiple views to support top-k queries. The original proposal of *PREFER* constructs multiple views, and at query time, the system picks the view whose weights are closest to the query weights to answer the query. This idea can also be applied to *AppRI*, such that we can use the proposed method to build multiple ranked views. We demonstrate our approach by showing how to build 3 ranked views.

Suppose the number of dimension is 3 (A_1,A_2 and A_3), and the weights associating with each dimension are w_1, w_2 and w_3. we can classify all query weights into 3 categories: (1) w_1 is the minimum weight ($\min(w_1, w_2, w_3) = w_1$); (2) w_2 is the minimum weight ($\min(w_1, w_2, w_3) = w_2$) and (3) w_3 is the minimum weight $\min(w_1, w_2, w_3) = w_3$. Each query will fall into one and only one category. For those queries in the first case, we can rewrite the weights as

$$(w_1, w_2 - w_1, w_3 - w_1) \tag{3}$$

and all weights are still non-negative. If the ranked view is built on the transformed data $(A_1 + A_2 + A_3, A_2, A_3)$ (*i.e.*, aggregate the values on dimensions A_2 and A_3 to A_1 for

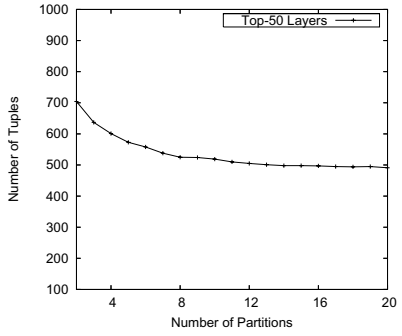

Figure 6: Number of Tuples w.r.t. Partition Number

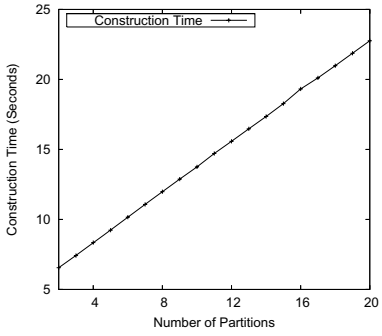

Figure 7: Construction Time w.r.t. Partition Number

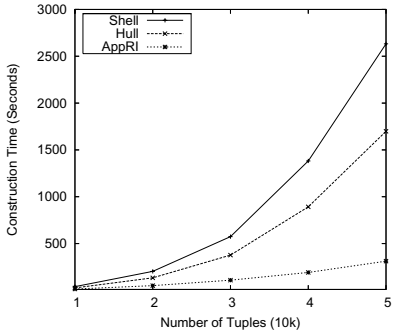

Figure 8: Construction Time w.r.t. Data Size

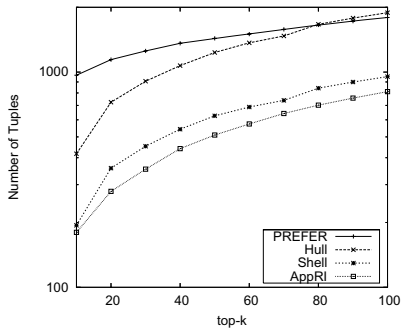

Figure 9: Query Performance on Uniform Data

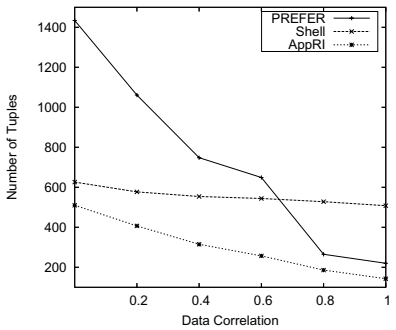

Figure 10: Query Performance on Correlated Data

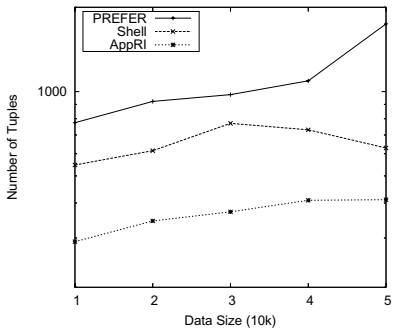

Figure 11: Query Performance w.r.t. Data Size

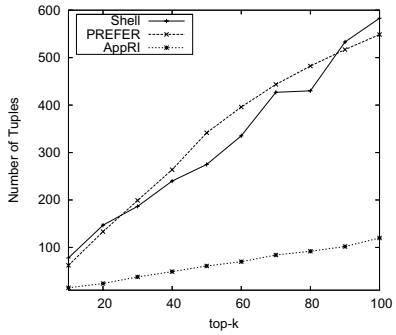

Figure 12: Query Performance on Abalone3D Data

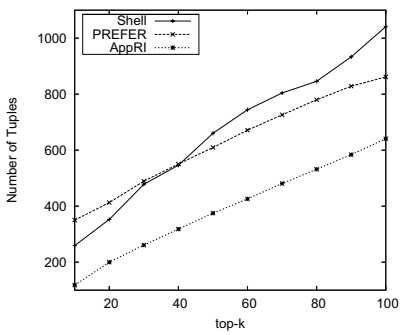

Figure 13: Query Performance on Cover3D Data

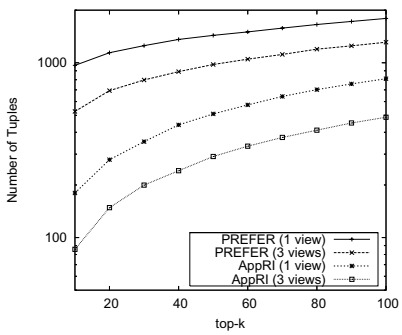

Figure 14: Query Performance on Multi-Views

each tuple), one can verify that the querying the rewritten weights on the transformed data can exactly answer the original query. Generally, we can conduct 3 transformation on the original data: $(A_1 + A_2 + A_3, A_2, A_3)$,$(A_1, A_1 + A_2 + A_3, A_3)$ and $(A_1, A_2, A_1 + A_2 + A_3)$, corresponding to the 3 different cases of the query weights classified above. For each transformed data, we use *AppRI* to build a ranked view. During query processing, we will first classify the query into the corresponding category (*i.e.*, whether w_1, w_2 or w_3 is minimal), then rewrite the query (*i.e.*, similar to Eqn. (3)) and use the associated ranked view to answer the query.

The idea can be generalized to m views. All possible query weights consists a multi-dimensional space, which can be divided into m subspaces. Each subspace corresponds to a linear transformation from the original space. The reversed linear transformation will be applied to the database to build a transformed data. We omit the detailed exploration in this paper.

We compare the query performance of *AppRI* and *PREFER* with 3 views, using the same data set in Figure 9. The *PREFER* system uses its own method to generate 3 views (*i.e.*, by ranking coverage). The average number of tuples is shown in Figure 14. Using 3 views, both *AppRI* and *PREFER* improve the query performance. With *AppRI*, we observe that the top k layers contain less number of tuples on the transformed data. This is because by aggregating A_2 and A_3 to A_1, the tuples are projected to a smaller subspace where A_1 must be larger than A_2 and A_3. As a result, more domination relations can be discovered.

7. DISCUSSION

We discuss the possible extensions of the proposed method.

7.1 Partial Indices

In many top-k queries, the value of k is relative small comparing with the data size. It is unnecessary to compute a full index which includes all layers. Instead, the top k layers are sufficient to correctly answer the queries. In the case where multiple views are used, building partial index can significantly reduce the space requirement. For example, in the synthetic data with $50k$ tuples, the number of tuples in the top-100 layers of *AppRI* is $1,377$. In a multi-view system, *AppRI* can build approximately 36 partial views to guarantee any top-100 queries, while the consumed space is equivalent to a single complete view. We expect the system with multi-views will have significant improvement over that with a single view, as shown in Section 6.4. This additional benefit is very limited for *Shell* because the number of tuples in the top k layers constructed by *Shell* is much larger than that by *AppRI*. For example, with the same data discussed above, the top-100 layers in *Shell* have $28,854$ tuples. Using the same space, *Shell* is only able to build at most 2 views for top-100 queries.

7.2 Non-Monotone Queries

Throughout the paper, we assume linear queries are monotone (*i.e.*, all weights $w_i \geq 0$). Generally, this assumption holds because people have preference on each attribute. However, sometimes different users may have different preferences (*i.e.*, user may issue either positive or negative weights). This situation can be handled by the same idea of multiple views discussed in Section 6.4. More specifically, for a d dimensional database, we have at most 2^d different cases on query weights (*i.e.*, either positive or negative) and we can build multiple indices for each case. Since a negative preference is the same as a positive preference on the negated values, the *AppRI* algorithm can be used without modification. For example, in a relation with $d = 3$, if the user have non-determined preference on attribute A_1. We will build indices for both (A_1, A_2, A_3) (original data) and $(-A_1, A_2, A_3)$ (transformed data).

7.3 Index Maintenance

Different from query processing, index maintenance (*i.e.*, inserting and deleting) on robust index is fairly complex. Updating is not discussed here because it can be seen as a deletion followed by an insertion. Because of the expensive computation, it may be advisable in practice to perform index maintenance in batches. We suggest two temporal solutions for online maintenance. For deletion, the tuple can be marked as deleted, but is not really removed from the database. At query time, if a marked tuple appears in top-k, the system needs to retrieve one more layer. For insertion, one can count the number of tuples which dominate the new tuple (this can be accomplished by issuing an SQL query). Suppose the count is n, the new tuple can be inserted into the $(n+1)^{th}$ layer. The index can be rebuilt periodically.

7.4 High Dimensional Data

In our experiments, we mainly compare different methods with 3 dimensions. Basically, all methods suffer from high dimensionality. Both *AppRI* and *Shell* will include more tuples in each layer since tuples are harder to dominate each other in higher dimensions. *PREFER* is also weakened because the space of possible query weights increases and the pre-computed index is more difficult to cover the queries. For high dimensional data, a unified index structure over all attributes may not be practical. A possible alternative is to combine the methods in distributive index [10]. We can construct low dimensional ranked views and answer high dimensional queries by merging some existing low dimensional views. There are several interesting issues in this direction: first, how to partition the dimensions to build the low dimensional views; and second, how to optimize a merge plan for high dimensional query using the existing views. We will further explore this direction as a future work.

8. CONCLUSIONS

To efficiently answer top-k queries, we proposed a new indexing criterion: robust index. We discussed the necessary and sufficient conditions for robust index and developed a practical method to approximate the exact solutions. Our experimental results show that the proposed approach outperforms the previous studies.

9. APPENDIX

Sketch of Proof for Theorem 2.

When $d > 2$, consider an arbitrary tuple $t \in R$, we first pick any other $d-2$ tuples $t'_1, t'_2, \ldots, t'_{d-2}$ and let $S = \{t, t'_1, t'_2, \ldots, t'_{d-2}\}$. For any tuple $v \in R - S$, $\{v\} \cup S$ construct a hyperplane (a line when $d = 2$). Similarly, we can sort $v \in R - S$ and compute the minimum ranking value for S. This procedure takes $O(n \log n)$ time . Since there

are $\binom{n-1}{d-2}$ different S's in total, the time complexity to compute minimum ranking for all $t \in R$ is $O(n\binom{n-1}{d-2}n\log n) = O(n^d \log n)$. ∎

Sketch of Proof for Theorem 3.
We first derive an upper bound for $l(t, L^*)$. In Figure 4, each partitioning line corresponds to a linear query (as shown in Example 1). Consider the line between I_1 and I_2. Let the corresponding query be q_1. The rank of t with respect to q_1 is the number of tuples in region II (*i.e.*, $|\mathcal{DS}^1(t)|$) and subregions I_1,III_1,$III_2, \ldots, III_{B-1}$. According to the definition of robust index, we have:

$$l(t, L^*) \leq |\mathcal{DS}^1(t)| + |I_1| + \sum_{i=1}^{B-1} |III_i|$$

By considering all partitioning lines, we can derive:

$$\begin{aligned}
l(t, L^*) &\leq |\mathcal{DS}^1(t)| + \min(|I_1| + \sum_{i=1}^{B-1} |III_i|, |I_1| + |I_2| \\
&+ \sum_{i=1}^{B-2} |III_i|, \ldots, \sum_{i=1}^{B-1} |I_i| + |III_1|) \\
&\leq |\mathcal{DS}^1(t)| + \min(|I_1| + \sum_{i=1}^{B-2} |III_i|, \ldots, \sum_{i=1}^{B-2} |I_i| + |III_1|) \\
&+ \max(|I_1|, \ldots, |I_{B-1}|, |III_1|, |III_{B-1}) \\
&= l(t, \hat{(L)}) + \max(|I_1|, \ldots, |I_{B-1}|, |III_1|, |III_{B-1})
\end{aligned}$$

Let $m = \max(|I_1|, \ldots, |I_{B-1}|, |III_1|, |III_{B-1})$ and $n = \min(|I_1| + \sum_{i=1}^{B-2} |III_i|, \ldots, \sum_{i=1}^{B-2} |I_i| + |III_1|)$. If the data is uniformly distributed, we have

$$E[\frac{m}{l(t, \hat{L})}] \leq E[\frac{m}{n}] = \frac{1}{B-1}$$

We conclude $E[\frac{l(t,\hat{L})}{l(t,L^*)}] \geq 1 - \frac{1}{B}$. ∎

Sketch of Proof for Lemma 4.
Since $\bigcup_{j=1}^{i} I_j = a_i$ and $\bigcup_{j=1}^{B-i} III_j = b_i$, it is equivalent to show that any tuple $t_a \in a_i$ can be paired with any tuple $t_b \in b_i$ to form a 2-dominating set of t. That is, we need to show $\exists v \in [0, 1]$ such that $vt_a + (1-v)t_b$ dominates t. Let $v_i (i = i_1, i_2, \ldots, i_l)$ be the weights such that $v_i t_a^i + (1 - v_i)t_b^i = t^i$. We select a v_{i^*} such that $i^* = \arg\max_{i=1}^{i_l} v_i$. Since $i_1, i_2, \ldots, i_l$ are dominating (dominated) dimensions of subspace a (b), we have $t_a^i \leq t^i \leq t_b^i, i = i_1, i_2, \ldots, i_l$. Thus,

$$v_{i^*} t_a^i + (1 - v_{i^*})t_b^i \leq t^i, i = i_1, i_2, \ldots, i_l$$

The next step is to show that for each $j = j_1, j_2, \ldots, j_g$,

$$v_{i^*} t_a^j + (1 - v_{i^*})t_b^j \leq t^j$$

This can be done by a simple calculation from two inequalities:

$$\begin{aligned}
\gamma_p t_a^{i^*} + t_a^j &\leq \gamma_p t^{i^*} + t^j \\
\gamma_p t_b^{i^*} + t_b^j &\leq \gamma_p t^{i^*} + t^j.
\end{aligned}$$

Multiply v_{i^*} to both sides of the first equation, and multiply $(1 - v_{i^*})$ to those of the second equation, then sum them up. We have

$$\begin{aligned}
&v_{i^*}\gamma_p t_a^{i^*} + v_{i^*} t_a^j + (1 - v_{i^*})\gamma_p t_b^{i^*} + (1 - v_{i^*})t_b^j \\
\leq &v_{i^*}\gamma_p t^{i^*} + v_{i^*} t^j + (1 - v_{i^*})\gamma_p t^{i^*} + (1 - v_{i^*})t^j \\
\Rightarrow &v_{i^*} t_a^j + (1 - v_{i^*})t_b^j \leq t^j
\end{aligned}$$

∎

10. REFERENCES

[1] US News and World Reports. *http://www.usnews.com/usnews/home.htm.*

[2] Pankaj K. Agarwal, Lars Arge, Jeff Erickson, Paolo Giulio Franciosa, and Jeffrey Scott Vitter. Efficient searching with linear constraints. *Proceedings of the 1998 ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'98)*, pages 169–178, 1998.

[3] C. Blake and C. Merz. UCI Machine Learning Repository. *http://www.ics.uci.edu/ mlearn/MLRepository.html.*

[4] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. *ICDE*, 2001.

[5] Y. Chang, L. Bergman, V. Castelli, M. Lo C. Li, and J. Smith. Onion technique: Indexing for linear optimization queries. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*, pages 391–402, 2000.

[6] Surajit Chaudhuri and Luis Gravano. Evaluating top-k selection queries. pages 397–410, 1999.

[7] T. Cormen, C. Leiserson, and et al. Introduction to algorithms. *The MIT Press*, 2001.

[8] R. Fagin. Combining fuzzy information from multiple systems. *Proceedings of the 1996 ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'96)*, pages 216–226, 1996.

[9] R. Fagin. Fuzzy queries in multimedia database systems. *Proceedings of the 1998 ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'98)*, pages 1–10, 1998.

[10] R. Fagin, A Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Proceedings of the 2001 ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'01)*, 2001.

[11] W. Ford and W. Topp. Data structures with c++. *Prentice-Hall*, 1996.

[12] J. Goldstain, R. Ramakrishnan, U. Shaft, and J. Yu. Processing queries by linear constraints. *Proceedings of the 1997 ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'97)*, pages 257–267, 1997.

[13] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD'01)*, pages 259–270, 2001.

[14] H. Kung, F. Luccio, and F. Preparata. On finding the maxima of a set of vectors. *J. of ACM*, 22, 1975.

[15] L. Lovasz and M. Plummer. Matching theory. *Amsterdam, Netherlands: North-Holland*, 1986.