Efficient Approximation of The Maximal Preference Scores by Lightweight Cubic Views

Yueguo Chen[1], Bin Cui[2], Xiaoyong Du[1,3], Anthony K. H. Tung[4]
[1]Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China), MOE, China
[2]Department of Computer Science, Peking University, Beijing, China
[3]School of Information, Renmin University of China, Beijing, China
[4]School of Computing, National University of Singapore, Singapore
chenyueguo@ruc.edu.cn

ABSTRACT

Given a multi-features data set, a best preference query (BPQ) computes the maximal preference score (MPS) that the tuples in the data set can achieve with respect to a preference function. BPQs are very useful in applications where users want to efficiently check whether many individual data sets contain tuples that are of interest to them. Although a BPQ can be naïvely answered by issuing a top-1 query and computing the score from the returned tuple, doing so might require to load a larger number of tuples externally. In this paper, we address the problem of efficient processing BPQs by using lightweight cubic (3-dimensional) views. With these in-memory views, the MPSs of BPQs can be efficiently estimated with an error bound guaranteed, by paying only a small number of I/Os. Extensive experimental results over real-life data sets show that our approximate solution can achieve the efficiency of up to three orders of magnitude compared to exact solutions, with certain accuracy guaranteed.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: System—*query processing*

General Terms

Algorithms, Experimentation, Performance

Keywords

preference query, best preference score, top-k query, materialized views

1. INTRODUCTION

Preference queries have been widely used for ranking objects with multiple features [3, 4, 5, 9, 13, 12]. They serve for retrieving a small set of tuples with top preference scores over multiple features, from a set of tuples. A tuple t of multiple (r) features is represented as a vector $\mathbf{t} \in \Re^r$, where $\mathbf{t^i}$ is the ith feature of t. Without loss of generality, we assume that high feature values are preferred by users for all the features. The kernel of preference queries is the scoring function $S(q,t)$ for measuring the preference score of a tuple t to a given preference query q. Monotonic scoring functions are typically applied in many existing studies on preference queries [4, 5, 9]. A preference function S is monotonic if for any given two tuples t_1 and t_2, $\mathbf{t_2^i} \geq \mathbf{t_1^i}$ for $i = 1, \ldots, d$, $S(q,t_2) \geq S(q,t_1)$ holds. Although there are many monotonic functions, the linear preference function, which is a simple linear product ($S(q,t) = \mathbf{q} \cdot \mathbf{t}$) of a preference vector $\mathbf{q} \in \Re^r$ ($\mathbf{q}^i \geq 0$) and a tuple $\mathbf{t}$, is more widely used [5, 7, 8, 13, 12, 17]. Because of the wide usage, we only consider the linear preference function in this work. In practice, a preference query typically has only a small number (e.g. ≤ 6, [5, 8, 13, 12, 15]) of features whose preference weights are larger than zero (called ranking features). In this paper, the dimensionality of a query q, d, actually refers to the number of ranking features of q. A query vector can be normalized as $\| \mathbf{q} \|_1 = 1$ (L_1-norm) without changing the order of tuple preference. We therefore use normalized query vectors in this study.

Given a data set T of tuples, a best preference query (BPQ) q, is to compute the maximal preference score (MPS) that can be achieved by tuples in T, i.e., to compute $S(q,T) = \sup_{t \in T} S(q,t)$. BPQ is very useful in applications where users want to efficiently check whether many individual data sets contain tuples that are of interest to them. For example, in a location-based service application, a driver presents a preference query to continuously monitor interesting commodities sold by surrounding shops. To avoid the frequent interruption of finding too many items, a high preference threshold may be used for pruning most items. In this case, the BPQ will be very useful for quickly pruning data sets whose MPSs are not large enough. In a slightly different example of online shopping application, a user presents a top-k query for retrieving global top-k commodities among a large number of data sources (which are online selected with some conditions). With BPQ, we can quickly prune those data sources that may not contain any commodities of the global top-k results. When we have n data sources and $k \ll n$, our method will be efficient to prune off most of the n data sources. That is, BPQ is useful and efficient in pruning a data set with a large probability that it does not contain any tuples preferred by the user.

EDBT 2012, March 26–30, 2012, Berlin, Germany.

A BPQ can be simply treated as a variant of a top-1 preference query, whose goal is varied as the preference score of the top-1 tuple, instead of the tuple itself in the top-1 preference query. As a variant of the top-k query, existing algorithms on top-k queries [3, 4, 9, 13, 19] can be simply adapted to address the BPQ. However, as we will show with the two widely used algorithms [9, 19] later, a large number of tuples need to be read externally for data sets of large skyline cardinality. As a result, the efficiency of BPQs is degraded due to the large number of I/Os incurred for loading tuples.

Given the motivating applications of BPQs mentioned above, a close approximation of the MPSs will not affect the effectiveness of BPQs significantly. A potential benefit of the approximation is the significant efficiency improvement that can be achieved. Although the dimensionality of a dataset can be as large as tens or even hundreds, practical BPQs usually contain no more than 6 ranking features [5, 8, 13, 15]. We observe that the MPSs of those low dimensional queries can be effectively approximated by using cubic views. Creating views of high dimensionality will be not economic for queries of low dimensionality. We therefore propose an approximate solution for computing the MPSs of low dimensional BPQs by 3-dimensional views. Instead of always giving an exact result of an MPS, our algorithm provides a tight range (which is a pair of a lower bound and an upper bound) that the MPS will lie in. We will show that by trading off on the accuracy, we can achieve the much desired efficiency on computing an MPS with a small error guaranteed. The main contributions of the paper are summarized as follows:

- We propose an effective mechanism for bounding the MPSs for practical BPQs. A number of lightweight cubic views are carefully materialized, for tightly bounding the MPSs of queries.
- We propose an indexing structure to index the partitioned query space, so that effective cubic views for bounding an ad hoc query can be efficiently discovered. The lightweight views allow the updates of views and indexes to be efficiently processed.
- An error bound of the MPS is always guaranteed. While it is designed as an approximate solution, the exact results are however often returned.
- Extensive experiments over real-life data sets demonstrate the efficiency of BPQs can be significantly improved by our solution.

The rest of the paper is organized as follows: Section 2 presents related work. Section 3 provides two baseline solutions of computing MPSs by extending existing top-k algorithms. Section 4 introduces the upper bound and lower bound of MPSs achieved by views. Section 5 presents the approximate solution on efficient query processing for BPQs. The experimental studies are described in Section 6. The conclusion is drawn in Section 7.

2. RELATED WORK

Top-k preference queries have been widely studied [3, 4, 5, 9, 13, 12, 19]. A top-k preference query is to retrieve the k highest scoring tuples from a given data set. Figure 1 gives an example in which the query vector $\mathbf{q}$ is represented as a line initiated from the origin and perpendicular to the plane $\Sigma_{i=1}^{d}\mathbf{q}^i\mathbf{t}^i = c$. All the tuples falling on the plane $\Sigma_{i=1}^{d}\mathbf{q}^i\mathbf{t}^i = c$ have the same projection on $\mathbf{q}$. They therefore have the same preference score c, which is proportional to the length of the projection of $\mathbf{t}$ on $\mathbf{q}$. Top-k tuples are therefore detected on those planes (perpendicular to $\mathbf{q}$) far from the origin.

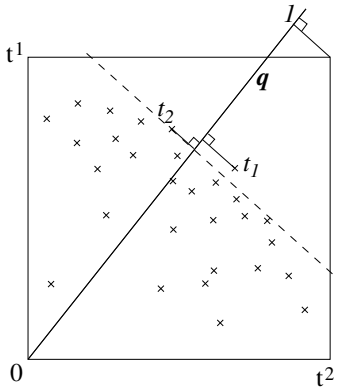


Figure 1: A top-2 preference query on 2D data set.

A good survey of top-k query processing techniques can be found in [14], with most top-k query algorithms assuming that the data are stored in external devices. Consequently, reducing the cost of sequential and random I/Os for reading the tuples is the main goal of many top-k algorithms. The main design consideration of these top-k algorithms is therefore to save the I/O cost by using the information precomputed from the order of tuples in different ways. One category of techniques on top-k queries use inverted lists to rank tuples for different features. Fagin et al. [9] proposed two well known algorithms: No Random Access (NRA) and Threshold Algorithm (TA). To retrieve top-k items, the inverted lists (one for each feature) are sequentially scanned in parallel, and since items in the inverted lists are ranked, the upper bound of interest scores of the unseen items can be easily computed. Once the score of the k^{th} item detected is no less than the upper bound, all the unseen items can be directly pruned.

The second category of top-k query processing techniques organize tuples of data sets in layers [5, 17, 19] based on the dominating relationships (or convex hull) among tuples. As a result, tuples with high preference scores are most likely in the outer layers. The algorithms achieve efficiency on top-k queries by pruning the I/O accesses for those tuples that are dominated by a small number of tuples which have been scanned. The dominating relationships of tuples, more popularly known as skyline queries, have been widely studied [2, 6, 10], and the cardinality of skylines was examined and studied in [11, 18].

In [8, 13, 12], a view is an inverted list created from the results of a specific top-k query and the TA algorithm [9] was adapted to scan high scoring tuples over the materialized views for the top-k queries. However, effective views creation has not been addressed. Approximate algorithms for top-k queries have been tried [1, 16]. These algorithms try to give a probabilistic evaluation of a newly seen item to be a top-k item or the seen top-k items to be the exact top-k results so

that top-k queries can be efficiently processed with certain accuracy guaranteed. The θ-approximation of top-k queries is also introduced to speedup the processing of top-k queries by early stopping [9]. It is guaranteed that the preference scores of tuples not among the top-k results will not be larger than θ ($\theta > 1$) times of those of top-k results. Although top-k query has been widely studied, efficient BPQ processing has never attracted much attention from researchers.

Table 1 gives the notations used in this paper:

Table 1: Frequently used notations

q, $\mathbf{q}$	a query and its vector representation
v, $\mathbf{v}$	a view and its vector representation
t, $\mathbf{t}$	a tuple and its vector representation
$\mathbf{q}^i$, $\mathbf{v}^i$, $\mathbf{t}^i$	the ith feature of a vector
d	the dimensionality of queries
$S(q,t)$	preference score of a tuple t for a query q
$S(q,T)$	the MPS of a data set T for a query q
$U(q,T)$	the upper bound of $S(q,T)$
$L(q,T)$	the lower bound of $S(q,T)$
$v.s$	the MPS of a view v
$v.t$	the outermost tuple of a view v
s	the exact maximal preference score
$\bar{s}$	the approximate maximal preference score
ε	the tolerance on the uncertainty of $S(q,T)$
w_i	a weight satisfying $0 \leq w_i \leq 1$
$g(v_1,\ldots,v_k)$	the variance of a convex
δ	maximal variance tolerable for a convex
h_{max}	maximal height of the index tree

3. BASELINE SOLUTIONS FOR COMPUTING THE MAXIMAL PREFERENCE SCORE

We first introduce two extended solutions of top-k algorithms.

3.1 The extended TA algorithm

The first algorithm is the extended θ-approximation of the TA algorithm (ETA, Algorithm 1) which is essentially a top-1 algorithm based on TA [9]. As the TA algorithm for processing top-k queries, inverted lists are sequentially scanned in parallel. The computation of the preference score of a scanned tuple requires a random I/O access. The approximate MPS $\bar{s}$ is updated when a higher score of a scanned tuple is detected. The scanning of inverted lists terminates when the preference scores of the unseen tuples cannot be larger than θ ($\theta \geq 1$) times of the $\bar{s}$ of the scanned tuples. It is guaranteed that the exact MPS s satisfies $\bar{s} \leq s \leq \theta\bar{s}$. The exact MPS is computed when $\theta = 1$. The cost of ETA is highly affected by the number of tuples that require random I/Os. We use the example in Figure 2(a) to illustrate this. Assuming we only have two dimensions and two sorted lists l_1 and l_2. When l_1 and l_2 are accessed, the minimal scanned score (b_1 and b_2) of each list is updated. The ETA algorithm terminates when $\Sigma_{i=1}^{d}(\mathbf{q}^i \cdot b_i) \leq \theta\bar{s}$ satisfies. As a result, all tuples within the square $OABC$ are pruned. Comparatively, each of the tuples in the shaded region requires one random I/O access to get its exact preference score. Therefore, although a large percentage of tuples can be pruned by ETA, there are still a certain number of tuples requiring random I/Os.

Algorithm 1 : The extended TA algorithm (ETA)

Input: $\mathbf{q}$, a preference weighting vector.
Output: $\bar{s}$, the approximate result of the exact MPS s.

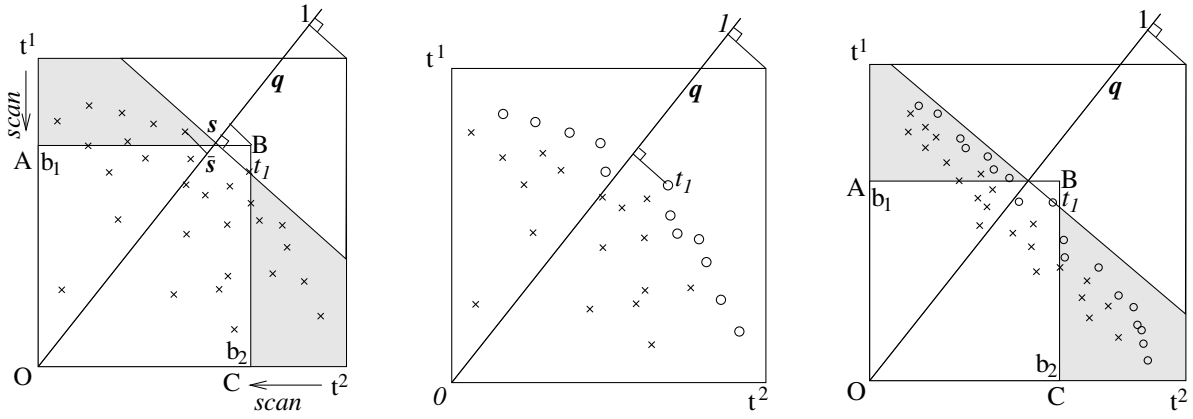
```
1. V = ∅ {records the tuples having been scanned}
2. s̄ = −∞.
3. let l_1, ..., l_d be the inverted lists for features 1, ..., d.
4. let t_1, ..., t_d be the current tuple scanned on each inverted list. {tuples are scanned from head to end}
5. let b_1, ..., b_d be the corresponding feature score of t_1, ..., t_d in each inverted list.
6. while Σ_{i=1}^{d}(q^i · b_i) > θs̄ do
7.   for i = 1, ..., d do
8.     if t_i ∉ V then
9.       compute S(q, t_i) by a random I/O access for t_i.
10.      insert t_i into V.
11.      if S(q, t_i) > s̄ then
12.        s̄ = S(q, t_i).
13.      end if
14.    end if
15.    scroll down the cursor t_i in l_i.
16.    update b_i be the feature score of t_i in l_i.
17.  end for
18.  if scan to the bottom of inverted lists then
19.    break.
20.  end if
21. end while
```

3.2 The skyline scan algorithm

The fact that a linear preference function is monotonic implies that for a given query q, the MPS must be derived by a tuple in the skyline of the data set. Therefore, the computation of $S(q,T)$ can be simplified as retrieving the highest preference score from the skyline tuples of the data set. We call such an approach as the Skyline Scan Algorithm (SSA). The skyline tuples are assumed to be pre-computed offline by applying some existing skyline computation algorithms [10]. They are labelled in the dataset. The computation of an MPS therefore requires many random I/O accesses for retrieving relevant features of the skyline tuples. The cost of the SSA algorithm will be highly dependent on the number of skyline tuples in the dataset. Figure 2(b) shows the skyline tuples (in circles) of the running example.

There are also some other alternatives of computing the MPSs. However, a certain number of random I/Os have to be paid for computing the MPSs if no in-memory views or indexes help to bounding the MPSs. The number of random I/Os for the proposed solutions is somehow correlated to the skyline cardinality of the data set. A large number of random I/Os will be required if the skyline cardinality is large.

Due to the expensive I/O cost, it will be a challenge for computing the MPSs in applications where a large number of BPQs need to be frequently processed. In this paper, we avoid the expensive I/O cost incurred when computing the MPSs by giving an approximate solution. In our approximate solution, we will return two bounds of $S(q,T)$, $(L(q,T), U(q,T))$ such that $L(q,T) \leq S(q,T) \leq U(q,T)$ and $\frac{U(q,T)-L(q,T)}{L(q,T)} \leq \varepsilon$, where ε is a small positive tolerance



(a) ETA (b) SSA (c) Anticorrelated features

Figure 2: Tuples that cannot be pruned (in shade region) by exact solutions.

restricting the uncertainty of $S(q,T)$. The exact $S(q,T)$ will be returned if $U(q,T) = L(q,T)$.

4. BOUNDING THE MAXIMAL PREFERENCE SCORES

When two query vectors are close to each other, the preference score of these two queries on the same tuple should also be quite close. As a result, the MPSs of these two query vectors will also be quite close. This motivates us to estimate the MPS of an ad hoc query q by some nearby query vectors which have been pre-computed. These sampled pre-computed query vectors are called views.

4.1 The upper bound of MPSs

A view v is a query vector. The MPS of the view v over a given data set T is $v.s = S(v,T)$. A view v also records the tuple that achieves the MPS for the query vector $\mathbf{v}$. Such a tuple is called the outermost tuple of the view, denoted as $v.t$. Therefore, $S(v, v.t) = S(v,T)$. If there are multiple tuples achieving the same MPS for a view v, we randomly choose one of them as the outermost tuple $v.t$. With the concept of views, we have the following lemma to upper bound the MPS of an ad hoc query vector q:

LEMMA 1. *Given a query q and k views $v_1, \ldots, v_k$, if $\mathbf{q} = \Sigma_{i=1}^{k}(w_i\mathbf{v_i})$ for a set of w_i, where $0 \le w_i \le 1$ and $\Sigma_{i=1}^{k} w_i = 1$, then we have $U(q,T) = \Sigma_{i=1}^{k}(w_i v_i.s) \ge S(q,T)$.*

Proof: For any tuple $t \in T$:

$$\begin{aligned} & U(q,T) = \Sigma_{i=1}^{k}(w_i v_i.s) \\ = \quad & \Sigma_{i=1}^{k}(w_i S(v_i,T)) \\ \ge \quad & \Sigma_{i=1}^{k}(w_i(\mathbf{v_i}\cdot\mathbf{t})) \\ = \quad & (\Sigma_{i=1}^{k}(w_i\mathbf{v_i}))\cdot\mathbf{t} = \mathbf{q}\cdot\mathbf{t} \\ Therefore, \quad & U(q,T) \ge \sup_{t\in T}\mathbf{q}\cdot\mathbf{t} = S(q,T)\square \end{aligned}$$

Intuitively, Lemma 1 says that if the query vector q is linearly spanned (or within the spanning space) by v_1, ..., v_k in the vector space with associated weights w_1, ..., w_k, then the MPS of q will also be upper bounded by the weighted average of the MPSs for v_1, ..., v_k (with associated weights being w_1, ..., w_k). In this paper, we call $v_1, \ldots, v_k$ (which form a convex) the composite views of q. This also means that the vector $\mathbf{q}$ is within the convex formed by views $\mathbf{v_1}, \ldots, \mathbf{v_k}$. The weights $w_1, \ldots, w_k$ can be computed by solving the linear equation as: $\Sigma_{i=1}^{k} w_i\mathbf{v_i} = \mathbf{q}$:

$$w_i = \frac{|\mathbf{v_1} \ \ldots \ \mathbf{v_{i-1}} \ \mathbf{q} \ \mathbf{v_{i+1}} \ \ldots \ \mathbf{v_k}|}{|\mathbf{v_1} \ \mathbf{v_2} \ \ldots \ \mathbf{v_k}|}, \quad (|A| = det(A))$$

4.2 The tightness of upper bounds

To derive an upper bound $U(q,T)$ from views based on Lemma 1, we have to guarantee that for any ad hoc query vector $\mathbf{q}$, we can find k views to span $\mathbf{q}$. A simple way to achieve this is to apply d orthogonal unit vectors as basic views, i.e., for the ith view v_i, $\mathbf{v_i}^i = 1$, and $\mathbf{v_i}^j = 0$ where $j \ne i$. With these basic views, given an ad hoc query q, $U(q,T)$ can simply be expressed as $U(q,T) = \Sigma_{i=1}^{d}(\mathbf{q}^i \cdot v_i.s)$.

We use an example shown in Table 2 to illustrate the use of orthogonal basic views. In this example, $\mathbf{v}_1 = (1,0,0)^T$, $\mathbf{v}_2 = (0,1,0)^T$, $\mathbf{v}_3 = (0,0,1)^T$. The data set T has 7 tuples t_1 to t_7. We have three materialized maximal interest scores $S(v_1,T) = 5.5$, $S(v_2,T) = 4.5$, $S(v_3,T) = 5.0$. Given a query vector $\mathbf{q} = (\frac{1}{3},\frac{1}{3},\frac{1}{3})^T$, we can compute $U(q,T) = \frac{1}{3}S(v_1,T)+\frac{1}{3}S(v_2,T)+\frac{1}{3}S(v_3,T) = 5.0$. However, $S(q,T) = 2.9$ which is achieved by t_6 (i.e., $S(q,t_6) = 2.9$). It is obvious that the derived upper bound score $U(q,T)$ is rather loose.

Table 2: An example of a data set

tuple	D_1	D_2	D_3
t_1	3.0	1.6	0.7
t_2	0.7	0.9	5.0
t_3	1.9	3.4	1.5
t_4	1.1	4.5	0.5
t_5	5.5	0.4	1.2
t_6	2.6	2.9	3.2
t_7	3.3	1.3	2.8

To estimate the MPSs more accurately, a tighter upper bound $U(q,T)$ is required. Suppose that we have another three views $\mathbf{v'_1} = (0.5,0.3,0.2)^T$, $\mathbf{v'_2} = (0.2,0.5,0.3)^T$, $\mathbf{v'_3} = (0.3,0.2,0.5)^T$, we should have $\mathbf{q} = \frac{1}{3}\mathbf{v'_1} + \frac{1}{3}\mathbf{v'_2} + \frac{1}{3}\mathbf{v'_3}$. By materializing the three views, we have $S(v'_1,T) = 3.11$,

$S(v_2', T) = 2.93$, $S(v_3', T) = 2.96$. Therefore, $U(q,T) = \frac{1}{3}S(v_1', T) + \frac{1}{3}S(v_2', T) + \frac{1}{3}S(v_3', T) = 3.0$, which is very close to $S(q,T) = 2.9$. From the above example, we can see that the tightness of the derived upper bound scores is highly dependent on the applied views. It is important to select effective views to linearly span a query q so that a tighter upper bound $U(q,T)$ can be achieved. We have the following lemma to help estimate the tightness of the derived $U(q,T)$:

LEMMA 2. *Given a query q and a view v, let $k(q,v)$ be $k(q,v) = \sup_i \frac{\mathbf{v}^i}{\mathbf{q}^i}$. Then, $U(q,T) \leq \Sigma_{i=1}^{k}(w_i k(q,v_i)) \cdot S(q,T)$.*

Proof: For any tuple $t \in T$:

$$\begin{aligned}
& S(v,t) = \mathbf{v} \cdot \mathbf{t} = \Sigma_{i=1}^{d}(\mathbf{v}^i \times \mathbf{t}^i) \\
\leq\ & \Sigma_{i=1}^{d}(k(q,v)\mathbf{q}^i \times \mathbf{t}^i) = k(q,v)\Sigma_{i=1}^{d}(\mathbf{q}^i \times \mathbf{t}^i) \\
=\ & k(q,v)(\mathbf{q} \cdot \mathbf{t}) = k(q,v)S(q,t) \\
Therefore,\ & S(v,T) \leq k(q,v)S(q,T) \\
& U(q,T) = \Sigma_{i=1}^{k}(w_i S(v_i,T)) \\
\leq\ & \Sigma_{i=1}^{k}(w_i k(q,v_i)S(q,T)) \\
=\ & \Sigma_{i=1}^{k}(w_i k(q,v_i)) \cdot S(q,T) \\
Therefore,\ & S(q,T) \leq U(q,T) \leq \Sigma_{i=1}^{k}(w_i k(q,v_i)) \cdot S(q,T) \square
\end{aligned}$$

With Lemma 2, we have the following two corollaries.

COROLLARY 1. *For a query q and a view v, $k(q,v) \geq 1$.*

Proof: $\| \mathbf{q} \|_1 = \| \mathbf{v} \|_1 = 1$. There must exist an i such that $\mathbf{v}^i \geq \mathbf{q}^i$, i.e., $\frac{\mathbf{v}^i}{\mathbf{q}^i} \geq 1$. Therefore, $k(q,v) \geq 1$. Obviously, $k(q,v) = 1$ only if $\mathbf{q} = \mathbf{v}$. $\square$

COROLLARY 2. *Given a query q and a view v, if there is another view v_1 in the line segment between q and v, i.e., $\mathbf{v_1} = \lambda\mathbf{v} + (1-\lambda)\mathbf{q}$ $(0 \leq \lambda \leq 1)$, we have $k(q,v_1) \leq k(q,v)$.*

Proof: Let $k(q,v_1) = \frac{\mathbf{v_1}^i}{\mathbf{q}^i}$. According to Corollary 1, $\mathbf{v}_1^i \geq \mathbf{q}^i$. Because $\mathbf{v}_1^i = \lambda\mathbf{v}^i + (1-\lambda)\mathbf{q}^i$, we have $\mathbf{v}^i \geq \mathbf{v}_1^i \geq \mathbf{q}^i$. Therefore,

$$\begin{aligned}
k(q,v_1) &= \frac{\mathbf{v_1}^i}{\mathbf{q}^i} = \frac{\lambda\mathbf{v}^i + (1-\lambda)\mathbf{q}^i}{\mathbf{q}^i} \\
&\leq \frac{\mathbf{v}^i}{\mathbf{q}^i} \leq k(q,v) \square
\end{aligned}$$

We use an example of 2-dimensional vectors in Figure 3 to illustrate how $k(q,v_1) \leq k(q,v)$ holds in Corollary 2. In this example, $k(q,v_1) = \frac{\|\mathbf{v_1}\|_1}{\|\mathbf{v_1'}\|_1} = \frac{1}{\|\mathbf{v_1'}\|_1}$, and $k(q,v) = \frac{\|\mathbf{v}\|_1}{\|\mathbf{v'}\|_1} = \frac{1}{\|\mathbf{v'}\|_1}$. Obviously, $\| \mathbf{v_1'} \|_1 \geq \| \mathbf{v'} \|_1$. Therefore, $k(q,v_1) \leq k(q,v)$. Based on Corollary 2, along the line segment between $\mathbf{v}$ and $\mathbf{q}$, the closer $\mathbf{v_1}$ to $\mathbf{q}$, the less $k(q,v_1)$. $k(q,v_1) = 1$ when $\mathbf{v_1} = \mathbf{q}$.

To achieve tighter upper bound score $U(q,T)$, lower $k(q,v_i)$ is desired so that $\Sigma_{i=1}^{k}(w_i k(q,v_i))$ could be closer to 1. We therefore need to apply those views closer to $\mathbf{q}$, which have lower $k(q,v_i)$ (according to the Corollary 2). When the composite views $v_1, \ldots, v_k$ are very close, they may have the same tuple t as their outermost tuple. In this case, we have the following lemma to guarantee that tuple t also achieves the MPS for any query vector $\mathbf{q}$ falling in the convex of $\mathbf{v_1}, \ldots, \mathbf{v_k}$. In such a case, $S(q,t)$ is an exact result for MPS.

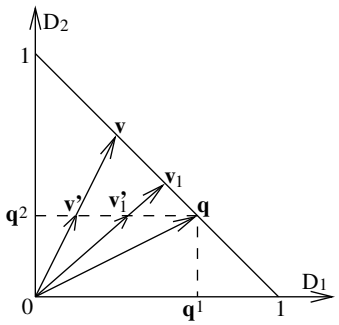


Figure 3: An example of bounding factors.

LEMMA 3. *If $\mathbf{q} = \Sigma_{i=1}^{k}(w_i\mathbf{v_i})$, where $0 \leq w_j \leq 1$ and $\Sigma_{i=1}^{k} w_i = 1$, and $v_1.t = v_2.t = \cdots = v_k.t = t$, then $U(q,T) = S(q,t) = S(q,T)$.*

Proof: We first prove $U(q,T) = S(q,t)$:

$$\begin{aligned}
& U(q,T) = \Sigma_{i=1}^{k}(w_i S(v_i,T)) \\
=\ & \Sigma_{i=1}^{k}(w_i S(v_i, v_i.t)) = \Sigma_{i=1}^{k}(w_i S(v_i,t)) \\
=\ & \Sigma_{i=1}^{k}(w_i \mathbf{v_i} \cdot \mathbf{t}) = \Sigma_{i=1}^{k}(w_i\mathbf{v_i}) \cdot \mathbf{t} \\
=\ & \mathbf{q} \cdot \mathbf{t} = S(q,t)
\end{aligned}$$

For any other tuple t' than t, because $v_i.t = t$, it is satisfied that $S(v_i,t') \leq S(v_i,t)$. Then we have,

$$\begin{aligned}
& S(q,t') = \mathbf{q} \cdot \mathbf{t}' = \Sigma_{i=1}^{k}(w_i\mathbf{v_i}) \cdot \mathbf{t}' \\
=\ & \Sigma_{i=1}^{k}(w_i(\mathbf{v_i} \cdot \mathbf{t}')) = \Sigma_{i=1}^{k}(w_i S(v_i,t')) \\
\leq\ & \Sigma_{i=1}^{k}(w_i S(v_i,t)) = \Sigma_{i=1}^{k}(w_i(\mathbf{v_i} \cdot \mathbf{t}) \\
=\ & \Sigma_{i=1}^{k}(w_i\mathbf{v_i}) \cdot \mathbf{t} = \mathbf{q} \cdot \mathbf{t} \\
=\ & S(q,t) \\
Therefore,\ & S(q,t) = S(q,T) \square
\end{aligned}$$

4.3 The lower bound of MPSs

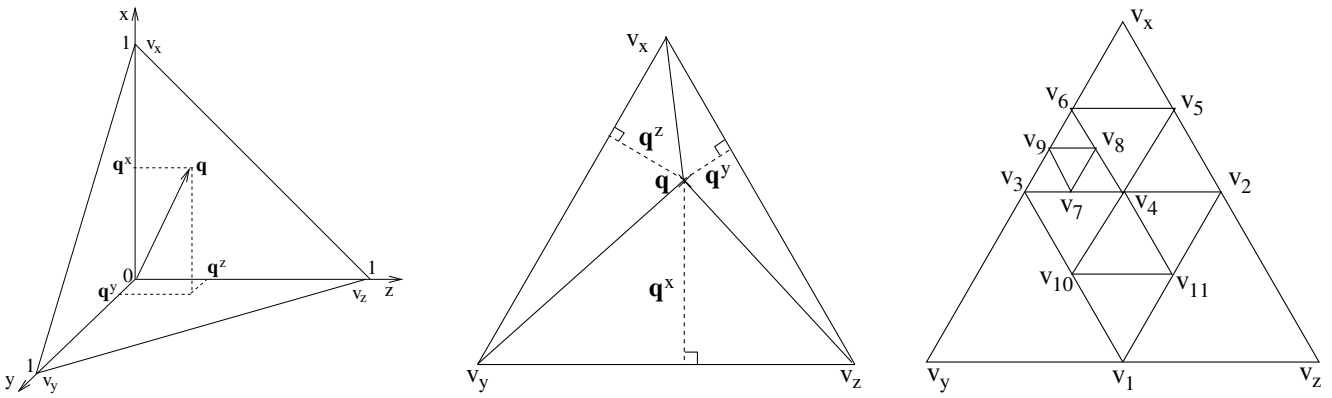
A lower bound of the MPS $S(q,T)$, $L(q,T)$, can be simply given by computing the preference score of a randomly selected tuple t from T. However, to provide tighter lower bound for $S(q,T)$, those outermost tuples of views that are near to $\mathbf{q}$ are preferred for computing the lower bound $L(q,T)$ because they are more likely to be the outermost tuples of q. In our solution, to maintain lightweight views (i.e., each view only maintains one outermost tuple achieving MPS on it), we simply derive the lower bound $L(q,T)$ from those outermost tuples of its composite views. For a query q and its composite views $v_1, \ldots, v_k$, the lower bound is computed as $L(q,T) = sup_{i=1}^{k} S(q, v_i.t)$.

4.4 Evaluating the tightness of the bounds

Views are important in determining the tightness of the upper bound score $U(q,T)$ and the lower bound score $L(q,T)$. Given k composite views $v_1, \ldots, v_k$ which form a convex, we define a measurement called the variance of a convex, denoted as $g(v_1, \ldots, v_k)$, to evaluate the tightness of the upper bounds and lower bounds for the MPSs of those query vectors falling into the convex:

$$g(v_1, \ldots, v_k) = \sup_{i=1}^{k} \sup_{j=1}^{k}(S(v_i,T) - S(v_i, v_j.t))$$

Lemma 4 estimates the tightness of $U(q,T)$ and $L(q,T)$:



(a) A query vector

(b) An alternative representation of the query vector

(c) Triangular partitions

Figure 4: The representation of 3 dimensional queries and views.

LEMMA 4. *Given a query q and its k composite views $v_1, \ldots, v_k$, $U(q,T) - L(q,T) \leq g(v_1, \ldots, v_k)$.*

Proof: Let $\mathbf{q} = \Sigma_{i=1}^{k}(w_i \mathbf{v_i})$. We have:

$$U(q,T) = \Sigma_{i=1}^{k}(w_i S(v_i,T)) = \Sigma_{i=1}^{k}(w_i S(v_i, v_i.t))$$

According to the definition of $L(q,T)$, there must be a $j \in \{1,\ldots,k\}$ such that:

$$\begin{aligned}
& L(q,T) = S(q,v_j.t) = \mathbf{q} \cdot v_j.\mathbf{t} \\
= \quad & \Sigma_{i=1}^{k}(w_i \mathbf{v_i} \cdot v_j.\mathbf{t}) = \Sigma_{i=1}^{k}(w_i S(v_i, v_j.t)) \\
Therefore, \quad & U(q,T) - L(q,T) \\
= \quad & \Sigma_{i=1}^{k}(w_i S(v_i, v_i.t)) - \Sigma_{i=1}^{k}(w_i S(v_i, v_j.t)) \\
= \quad & \Sigma_{i=1}^{k}(w_i (S(v_i, v_i.t) - S(v_i, v_j.t)) \\
\leq \quad & \Sigma_{i=1}^{k}(w_i g(v_1,\ldots,v_k)) = g(v_1,\ldots,v_k) \square
\end{aligned}$$

The variance of a convex $g(v_1,\ldots,v_k)$ can therefore help us to evaluate whether the composite views $v_1,\ldots,v_k$ are accurate enough in approximating $S(q,T)$, for any query vector $\mathbf{q}$ falling into the convex. If it is not enough, a further partition of the convex may have to be conducted.

5. APPROXIMATE THE MAXIMAL PREFERENCE SCORES BY CUBIC VIEWS

A number of views need to be created for effectively bounding the MPSs. These views partition the query space ($\| q \|_1 = 1$) into convexes. We propose to address an ad hoc query q by using the composite views of the convex which $\mathbf{q}$ falls on. In our study, 3-dimensional views are created for bounding the MPSs for some reasons: 1) the three dimensional query space provides an elegant partition strategy (that cannot be achieved in higher dimensional space) so that the partitions can be effectively indexed and retrieved; 2) many preference queries may simply focus on a small number of features, the views of more than three dimensions will be redundant for queries of less than 4 dimensions; 3) our solution is designed for practical BPQs of less than 6 ranking features, which can be simply addressed by cubic views. As such, we first mainly discuss query processing techniques on queries of no more than three ranking features.

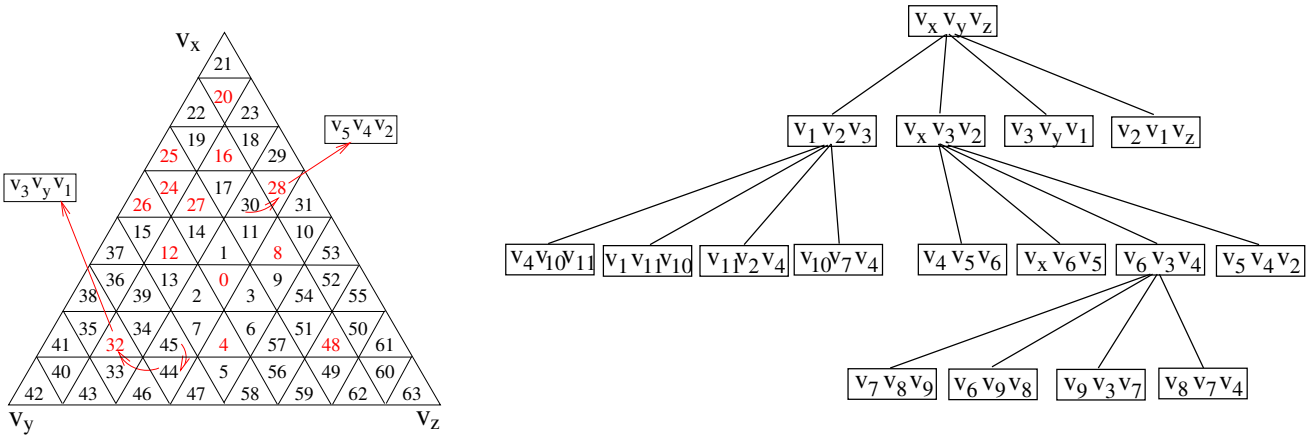
5.1 A simple representation of queries

Because all queries and views are normalized (i.e., $\| \mathbf{q} \|_1 = 1$), the queries and views will be within a plane of an equilateral triangle in 3-dimensional space. An example of a query vector with 3 features (x, y and z) is shown in Figure 4(a). As it is shown in Figure 4(b), a 3-dimensional query vector can be simply represented by an equilateral triangle in 2-dimensional space. A feature value of the query q (e.g., $\mathbf{q}^x$) can be represented as the distance of the query point to the corresponding edge (e.g., $v_y v_z$ for feature x) of the feature, with the height of the triangle (i.e., the distance of v_x to the edge $v_y v_z$) normalized as 1. In this way, as long as q is within the equilateral triangle $v_x v_y v_z$, $\mathbf{q}^x + \mathbf{q}^y + \mathbf{q}^z = 1$ always holds, because the sum of the areas of qv_xv_y, qv_yv_z, qv_zv_x is exactly the area of $v_xv_yv_z$. Because of the simplification in 2-dimensional representation, we present queries and views directly over the 2-dimensional equilateral triangle, where the triangle $v_xv_yv_z$ actually defines the space for all possible queries. The goal is to partition $v_xv_yv_z$ into small partitions by creating views so that queries can be effectively bounded by some composite views.

5.2 Partitioning and indexing the query space using cubic views

In our solution, we partition an equilateral triangle at the three middle points of the three edges, so that the big triangle is partitioned into four equal-sized equilateral triangles. An example is shown in Figure 4(c), where the triangle $v_xv_yv_z$ is divided into four equilateral triangles $v_xv_3v_2$, $v_3v_yv_1$, $v_2v_1v_z$ and $v_1v_2v_3$ by creating three views v_1, v_2, v_3. A partitioned triangle can be further partitioned to generate partitions of finer granularity. In such a way, the whole query space is hierarchically partitioned into a number of small equilateral triangles, as the example of Figure 4(c). Along with the partition of the query space, a 4-way tree index is built to index those partitioned triangles. An example of the tree index for the partition of Figure 4(c) is shown in Figure 5(b). Each node in the tree records the three vertex views. A non-leaf node also has links to its four child nodes.

However, generating a large number of fine granular partitions will increase the number of views as well as the maintenance cost of the views and the index structure. Many redundant partitions will be created if they are not well con-



(a) Ordering and search

(b) The tree index of partitions

Figure 5: Query space partitioning and partition search.

trolled. We have two ways of controlling the granularity of partitions (details are given in Algorithm 2). A parameter δ defines the desired variance for a leaf partition. We therefore do not further partition a triangle if the variance (function $g()$ defined in Section 4.4) of the triangle is no more than δ. However, considering that the variance of some fine granular partitions may still be larger than δ, we therefore also control the finest granularity of the partitions by constraining the maximal height of the tree index as h_{max}. Consequently, the size (i.e., width) of the finest granular partition would be the $1/2^{h_{max}}$ of that of the query space, while the volume would be $1/4^{h_{max}}$ of that of the query space. Therefore, the space cost for views and tree index is $O(4^{h_{max}})$. The complexity of creating indexes is also bounded as $O(4^{h_{max}})$, which is not important as the indexes are built offline.

5.3 View-based approximate BPQ processing

An important step of the approximate BPQ processing is to find the composite views for a given query q. We therefore need to search the tree index of partitions to find the leaf node whose partition contains the query vector $\mathbf{q}$. A straightforward way of finding such a leaf node is to trace the nodes from the root of the tree. However, the top-down approach of tracing nodes requires much comparison of vectors. We propose a bottom-up solution for finding the composite views of a query. We partition the whole space into equilateral triangles with the highest resolution (determined by h_{max}). An example is shown in Figure 5(a) where $h_{max} = 3$. As a result, there are totally $4^{h_{max}}$ cells. Those cells are ordered using a space filling curve. If we create a full index tree for all these highest resolution cells, the order of the cells in the filling curve will be exactly the order of leaf nodes in the full index tree. With such a filling curve, a leaf node in the index tree will correspond to an ordered cell in the fully partitioned space. For example, node $v_8v_4v_7$ corresponds to the ordered cell 27, and node $v_3v_yv_1$ corresponds to the ordered cell 32 in Figure 5(a). An ordered cell records a pointer if it corresponds to a leaf node of the index. For the tree shown in Figure 5, all the corresponding ordered cells of leaf nodes are colored in red in Figure 5(a).

By using orders, we actually create links from the space filling curve to the leaf nodes of the index tree. Given a query, we can efficiently look up the ordered cell c that $\mathbf{q}$ falls in based on its coordinates. If cell c has a pointer to a leaf node, we have found the desired partition in the index. Otherwise, we jump from cell c to another ordered cell which corresponds to the parent of the partition (in the tree index) corresponding to cell c. In this way, it is guaranteed that within h_{max} jumps we can find a leaf node in the tree index whose partition encloses $\mathbf{q}$. For example, a query falls in a cell 30 in Figure 5(a). Since the cell 30 has no pointer to a leaf node, it jumps to cell 28 which has a pointer to the leaf node $v_5v_4v_2$ in the tree index of Figure 5.

Algorithm 3 describes the details of view-based approximate BPQ processing (VBP), which is very efficient by limiting the complexity as $O(h_{max})$. Once a leaf partition containing the query vector is detected, the upper bound $U(q,T)$ and lower bound $L(q,T)$ are computed based on the composite views of the partition. A scan of the skyline tuples of T is required only if the relative error rate $\frac{U(q,T)-L(q,T)}{L(q,T)} > \varepsilon$, which may happen in those leaf partitions having a large variance.

5.4 Lightweight view maintenance

A view in the VBP solution only maintains the outermost tuple and its preference score on that view. Compared to the views used in LPTA [8] (top-k results), the views in VBP are more lightweight and therefore easy for maintenance. The updates of tuples in the data set may cause the materialized views to be updated. They therefore should be addressed. The updates of views may change the variance of some leaf nodes. This may result in the leaf node to be not tight enough for deriving bounds. It therefore need to be updated in two ways: 1) further partition on the node (to achieve finer granular partitions); 2) combine sibling nodes (to reduce the number leaf partitions). We only consider two cases of tuple updates: insertion and deletion. An update of a tuple can be treated as a delete operation followed by an insert operation.

5.4.1 *Tuple insertion*

The insertion of a tuple may enlarge the MPS of a view. Algorithm 4 is designed to address the tuple insertion. We first compare the MPS u that can be achieved by the inserted

Algorithm 2 Create a tree index for the query space

Input: T, the given data set.
Input: x, y, z, three features for creating views.
Input: δ, the largest variance allowed for a leaf partition.
Input: h_{max}, the maximal height of the tree index.
Output: r, the root of the tree index.
Output: V, the set of materialized views.

1. $V = \emptyset$
2. let v_x, v_y, v_z be the basic views for features x, y and z.
3. materializeView(v_x)
4. materializeView(v_y)
5. materializeView(v_z)
6. return r = newNode(v_x, v_y, v_z, 0)

newNode$(v_1, v_2, v_3, height)$:

1. n, initialize the new node.
2. **for** $i = 1$ to 3 **do**
3. $n.v_i = v_i$
4. **end for**
5. $n.g = g(v_1, v_2, v_3)$
6. $n.h = height$
7. **if** $n.h < h_{max}$ and $n.g > \delta$ **then**
8. partitionNode(n)
9. **end if**
10. return n

partitionNode(n):

1. materializeView($v_4 = \frac{n.v_2 + n.v_3}{2}$)
2. materializeView($v_5 = \frac{n.v_1 + n.v_3}{2}$)
3. materializeView($v_6 = \frac{n.v_1 + n.v_2}{2}$)
4. n.addAChild(newNode(v_4, v_5, v_6, $n.h + 1$))
5. n.addAChild(newNode($n.v_1$, v_6, v_5, $n.h + 1$))
6. n.addAChild(newNode(v_6, $n.v_2$, v_4, $n.h + 1$))
7. n.addAChild(newNode(v_5, v_4, $n.v_3$, $n.h + 1$))

materializeView(v):

1. **if** $v \notin V$ **then**
2. materialize v by computing $v.s = S(v, T)$.
3. let $v.t$ be outermost tuple for the view v.
4. insert v into V
5. **end if**

tuple with the minimal MPS b of all views. Those views are further checked only if $u > b$. The affected leaf nodes of the updated views will be recorded. They may be further updated based on the their updated variance, to keep the effective granularity of space partitions.

5.4.2 Tuple deletion

The deletion of tuples may reduce the MPS of a view. Since the tuple achieving the MPS for each view has been recorded, when a tuple is deleted, the affected views can be easily found. For an affected view v, an update of $v.s$ requires an expensive scanning of all the tuples. However, the original MPS of the view can still be applied as an upper bound of the MPS of the view. Therefore, the update of v can be delayed. If a tuple t with $S(v, t) \geq v.s$ is inserted after some delayed operations, the outermost tuple will be updated as t, so that the delayed updates of v can be directly ignored because they will not affect the updated $v.s$. However, when a certain number of tuples have been removed from the data sets, many original MPSs of views will be inaccurate. As a result, the derived bounds of MPSs will be not sufficiently tight. We therefore need to maintain a memo recording those delayed deletions, so that they can be processed in batch by only scanning tuples of data set once. The updates for the tree indexes in delete operations will be the same as those in insert operations.

Algorithm 3 View-based approximate BPQ processing (VBP)

Input: T, the given data set.
Input: $\mathbf{q}$, the query vector of at most 3 dimensions.
Input: r, the root of the tree index.
Output: $U(q, T), L(q, T)$

1. $X = \lfloor \frac{\mathbf{q}^x}{2^{h_{max}}} \rfloor$, $Y = \lfloor \frac{\mathbf{q}^y}{2^{h_{max}}} \rfloor$, $Z = \lfloor \frac{\mathbf{q}^z}{2^{h_{max}}} \rfloor$
2. look up a table for the entry (X, Y, Z) to get the order c of the partition that $\mathbf{q}$ falls in.
3. $h = 1$ {the level of the searched partition}
4. **while** $h <= h_{max}$ **do**
5. **if** $c.hasPointer$ **then**
6. $n = c.pointer$ {the node c points to}
7. **if** n does exist **then**
8. use $n.v_1$, $n.v_2$ and $n.v_3$ as the composite views of q
9. compute $U(q, T)$ and $L(q, T)$ from these views
10. **if** $\frac{U(q,T) - L(q,T)}{L(q,T)} \leq \varepsilon$ **then**
11. return $(U(q, T), L(q, T))$
12. **else**
13. compute $S(q, T)$ by scanning the skyline of T. {The same as the SSA.}
14. return $U(q, T) = L(q, T) = S(q, T)$
15. **end if**
16. break
17. **end if**
18. **else**
19. $c = c - c$ **MOD** 4^{h++}
20. **end if**
21. **end while**

5.5 Supporting moderate dimensional query

The techniques introduced in Algorithm 3 are focused on queries of no more than 3 ranking features ($d \leq 3$). However, it can be extended to support queries of moderate dimensions (e.g., $d \leq 6$) by sacrificing the pruning performance. To achieve this, we need to first rank the weights of all query features, and transform the query q into two sub-queries of no more than three features. The first three largest features form a feature group called the primary feature group (q_1 in Algorithm 5). The others (no more than 3 features) form the secondary feature group (q_2). For each of the two feature groups, a decomposed query is generated and normalized. The Algorithm 3 can then be applied to the decomposed queries to find the upper bound scores of these queries, from which the $U(q, T)$ can be finally evaluated by the weighted sum of the two upper bound scores. Algorithm 5 is designed to support queries of moderate dimensions. In terms of computational complexity, it can be reduced to Algorithm 3 which has a complexity of $O(h_{max})$. However, the pruning power of queries of moderate dimensions will be not as good as that of no more than 3 dimensions. To support queries of arbitrary features, the system need to materialize

Algorithm 4 Tuple insertion

Input: T, the data set T
Input: V, the set of views
Input: t, the inserted tuple

1. $N = \emptyset$, records leaf nodes affected
2. let b be the minimal maximal preference score of all views in V over data set T.
3. $u = \frac{(t^x)^2+(t^y)^2+(t^z)^2}{t^x+t^y+t^z}$.{The maximal preference score can be achieved by t.}
4. **if** $u > b$ **then**
5. **for** each view v **do**
6. **if** $S(v,t) > v.s$ **then**
7. $v.s = S(v,t)$, $v.t = t$
8. **for** each leaf node n using v as its composite view **do**
9. insert n into N
10. **end for**
11. **end if**
12. **end for**
13. **end if**
14. **for** each node $n \in N$ **do**
15. update $n.g$
16. **if** $n.g > \delta$ and $n.h < h_{max}$ **then**
17. further partition n based on Algorithm 2.
18. **else**
19. let n' be the parent of n, update $n'.g$
20. **if** $n'.g \leq \delta$ **then**
21. clear pointers of the ordered cells pointing to n''s children
22. remove the children of n'
23. make n' as a leaf node. Create a pointer for the ordered cell corresponding to the node n', pointing to n'.
24. **end if**
25. **end if**
26. **end for**

views and indexes for all possible subsets of the data sets' features with cardinality 3. Supporting preference queries of high dimensions is however beyond the scope of the techniques discussed in this paper.

6. PERFORMANCE EVALUATION

6.1 Experimental settings

We compare the performance of the VBP approach with those of four baseline solutions: θ-ETA, ETA, SSA and LPTA [8]. Among them, ETA, SSA and LPTA compute the exact MPS. For θ-ETA, we compute the approximation of MPS (where $\theta > 1$ in Algorithm 1). We use two real-life data sets: Household [1] and El Nino [2]. The Household data set (used in [18]) contains 127K tuples. Each tuple has 6 attributes recording the percentage of an American family's annual income spent on: gas, electricity, water, heating, insurance, and property tax. The El Nino data set contains oceanographic and surface meteorological readings taken from a series of buoys positioned throughout the equatorial Pacific. For the used data sets, we did a normalization and transformation based on the domain of the features such that larger values are considered better after the transformation. For scalability tests, data sets of different sizes are generated from the prefix of the whole data set. The query sets are randomly generated in the normalized query space. The testing results are based on the average of the results over 1000 queries.

[1] available at www.ipums.org
[2] available at kdd.ics.uci.edu/databases/el_nino/el_nino.html

Algorithm 5 VBP for queries of moderate dimensions

Input: T, the given data set.
Input: **q**, the query vector.
Input: R, the set of roots of the tree indexes.
Output: $U(q,T), L(q,T)$

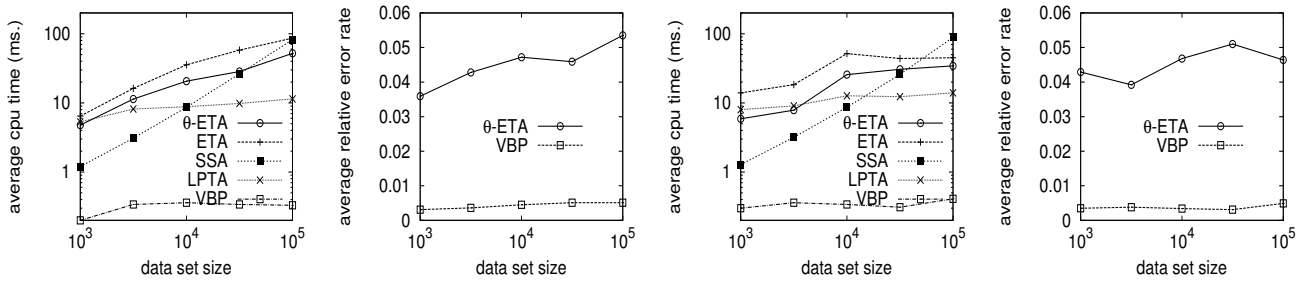
1. Rank all the features of **q** by their weights in the descending order.
2. Decompose the ranked features into two groups. Let the weights of the two groups be α_1 and α_2, which are the sum of the weights of their features respectively.
3. Normalize the two feature groups as q_1 and q_2.
4. find the composite views for q_1 and q_2 based on R.
5. $L(q,T) = \sup_i S(q, v_i)$, where v_i is a composite view.
6. $U(q,T) = \alpha_1 U(q_1,T) + \alpha_2 U(q_2,T)$
7. **if** $\frac{U(q,T)-L(q,T)}{L(q,T)} \leq \varepsilon$ **then**
8. return $(U(q,T), L(q,T))$
9. **else**
10. compute $S(q,T)$ by scanning over the skyline of T
11. return $U(q,T) = L(q,T) = S(q,T)$
12. **end if**

The data are stored in relational tables (MySQL system). They are retrieved through standard data access by SQL clauses. We therefore do not count I/Os, but the overall query execution time (note that the performance of SSA can benefit from hot cache). Skyline tuples of data sets are pre-computed and labeled by an additional attribute in relational tables. They are retrieved by SSA using SQL clauses like "select * from dataset where isskyline=1". Our experiments were conducted on a PC with a duo-core Intel (1.8GHz) processor and 3GB RAM. We implemented all the algorithms in C++. The database we use for storing and retrieving data is MySQL 5.0.

6.2 Performance of query processing

We first compare the performance of the approaches using a query set of 3 dimension. Figure 6(a) and 6(b) respectively give the results of efficiency and accuracy for the Household data set. Correspondingly, Figure 6(c) and 6(d) give the results for the El Nino data set. Because ETA, SSA and LPTA compute the exact MPSs, they therefore do not appear in Figure 6(b) and 6(d) for accuracy. From the results, we can see that the VBP approach is faster than the other approaches in one to three orders of magnitude. The performance of the other approaches suffer when the data sets are enlarged because large data sets typically have large skyline cardinality. This can be easily observed from the SSA approach. Comparatively, the performance of the VBP approach does not suffer significantly when the data set is enlarged mainly because the tight bounds derived from views help to prune most I/O accesses for skyline tuples.

As shown in Figure 6(b) and 6(d), the accuracy of the VBP approach is very good, with the average error rates are controlled under 0.005. Comparatively, the average error rates for the θ-ETA can be as large as 0.05, which is



(a) Efficiency, Household (b) Accuracy, Household (c) Efficiency, El Nino (d) Accuracy, El Nino

Figure 6: Performance comparison of best preference queries

much larger than that of VBP. We can also find from Figure 6(a) and 6(c) that, due to the approximation for computing MPSs, the θ-ETA approach is faster than the ETA approach. Similarly, the VBP approach is faster than the SSA approach. There is no significant difference between the results of the two test data sets.

In the experiments of Figure 6, the parameter ε of the VBP approach is set as a default value of 0.05. Note that the parameter ε will affect both the efficiency and the accuracy of the VBP approach. We study the trade off of efficiency and accuracy by adjusting ε for a number of values: 0.01, 0.02, 0.05 and 0.1. The impacts on the efficiency and the accuracy are shown in Figure 7. As we can see (Figure 7(a) and 7(c)), with the enlargement of ε from 0.01 to 0.1, the computational cost drops. However, the computational costs change slightly for $\varepsilon = 0.05$ and $\varepsilon = 0.1$ because almost all the queries can be effectively approximated when ε is large enough. In contrast, when $\varepsilon = 0.01$, the computational cost increases with the enlargement of the data set simply because the skyline cardinality is also enlarged. For accuracy (Figure 7(b) and 7(d)), obviously, the smaller the ε, the less the average error rates. Even though ε is as large as 0.1, the average error rates are still less than 0.015 in the tests of both data sets.

Parameters h_{max} and δ (used in Algorithm 2) for creating views and indexes affect the granularity of the query space partitioning. They therefore affect the tightness of the bounds achieved by the VBP approach. In the above experiments, we use the default values of these two parameters as $h_{max} = 3$ and $\delta = 0.05$. We further test the impacts of these two parameters on the VBP approach by only using the Household data set. The average error rate and CPU time of the VBP approach under various parameter settings are shown in Table 3. In general, the enlargement of h_{max} generates more views, which helps to improve the accuracy because finer partitions are achieved. The reduction of δ can generate more views. This is only verified in Table 3 when $h_{max} = 4$ because of the number of views are mainly controlled by h_{max} when h_{max} is small. We can also see that the accuracy increases (the error rate drops and the percentage of exact results increases) when more views are generated. More interestingly, the computational cost drops when h_{max} is enlarged. This is because less I/O accesses are required for retrieving skyline tuples when queries can be more effectively approximated.

6.3 Performance of updates

The update of a tuple may trigger the updates of some views and indexes. In our tests, for the deletions of tuples, the most pessimistic updating strategy is applied, i.e., once an outermost tuple of a view is deleted, we re-materialize the views by scanning all the tuples in the disk. The average updating cost (a delete operation plus an insert operation) over different size of data sets is shown in Table 4. We can see that when the size of data sets is enlarged, the chance of the re-materialization of views drops because the probability of a deleted tuple being the outermost tuple of a view also drops. However, the cost of one re-materialization process increases with the enlargement of data set size. That explains the phenomenon that the average update time increases and then drops with the enlargement of the data set size.

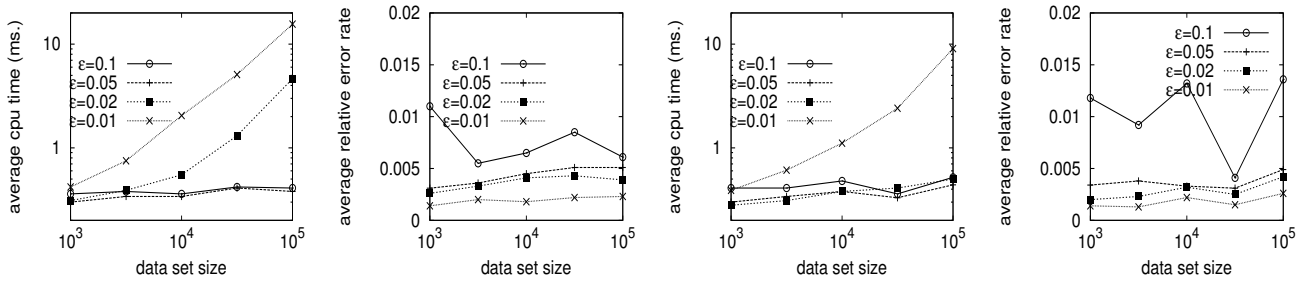
Table 3: The impact of h_{max} and δ on the performance of the VBP ($\varepsilon = 0.02$)

h_{max}	δ	views	err rate	exact results	time (ms.)
2	0.05	33	0.0041	17.6%	4.64
	0.02	33	0.0041	17.6%	4.63
	0.01	33	0.0041	17.6%	4.63
3	0.05	71	0.0034	41.3%	0.55
	0.02	71	0.0034	41.3%	0.53
	0.01	71	0.0034	41.3%	0.55
4	0.05	140	0.0020	57.9%	0.31
	0.02	163	0.0011	63.7%	0.31
	0.01	163	0.0011	63.7%	0.30

Table 4: The impact of the size of data sets on the update efficiency of the VBP approach.

data set (number of tuples)	average update time (ms.)	the ratio of re-materialization
1000	0.815	0.0922
3162	2.545	0.0771
10000	4.469	0.0535
31620	7.309	0.0387
100000	2.294	0.0015

The updating cost is also affected by the number of views maintained. We evaluate such an impact in Table 5. When the number of dimensions of data sets is increased, the number of views to be created increases. As a result, the average updating cost of tuples increases accordingly. This is because the probability of a deleted tuple being the outermost tuple of a view is increased. However, if the optimistic updating strategy is applied for the delete operations, the



(a) Efficiency, Household (b) Accuracy, Household (c) Efficiency, El Nino (d) Accuracy, El Nino

Figure 7: The trade off on efficiency and accuracy for the VBP approach

update cost will be much smaller (compared to the results of Table 5) because no re-materialization is required for updating the outermost tuples of those affected views. Those affected views can be efficiently addressed by some succeeding insert operations.

Table 5: The impact of the number of views on the update efficiency of the VBP approach

data set dimensionality	total number of views	avrg update time (ms.)	the ratio of re-material.
3	71	0.52	0.0061
4	193	1.44	0.0162
5	756	2.71	0.0228
6	989	5.30	0.0409

6.4 Queries of moderate dimensions

Many studies [5, 8, 13, 15] of preference queries exam the performance of their solutions using queries of less than 6 (although 2-3 dimensions are preferred) ranking features, which are supposed to take the majority of practical preference queries. We test the performance of the VBP approach (Algorithm 5) for queries of moderate dimensions. We apply a query set of 4 dimensions and a query set of 5 dimensions. For the θ-ETA approach, we set $\theta = 1.1$. For the LPTA approach, we evenly created views in the whole query space (5^4 views for the query set of 4 dimensions, and 4^5 views for the query set of 5 dimensions). Like the test for 3 dimensions, the number of views used for LPTA is larger than that used for the VBP approach. For the VBP approach, we set $\varepsilon = 0.05$ for the query set of 4 dimensions and $\varepsilon = 0.1$ for the query set of 5 dimensions. The results for the two query sets are given in Figure 8 and Figure 9 respectively.

As can be seen from Figure 8(a) and 8(c) for the query set of 4 dimensions and Figure 9(a) and 9(c) for the query set of 5 dimensions, the VBP approach outperforms the others when the data set is not significantly large. The efficiency of the VBP approach is surpassed by the LPTA approach (whose cost is not sensitive to the size of data set) for two tests when the data set has 100k tuples. Obviously, the efficiency of the VBP and SSA approaches is affected by the data set size, which actually affects the skyline cardinality. Comparatively, the VBP approach works relatively better for the El Nino data set than for the Household data set. For the accuracy, the VBP approach can achieve an average error rate of less than 0.02 for the query set of 4 dimensions and 0.04 for the query set of 5 dimensions, which is much better than that can be achieved by the θ-ETA approach.

7. CONCLUSION

In this paper, we have proposed a special preference query called the best preference query, and a very efficient approximate solution to efficiently process such a query. Given a BPQ, the tight upper bound and lower bound of the MPS can be efficiently derived from a few number of lightweight cubic views surrounding the query vector. We designed a space partitioning strategy and indexing structure for the query space so that effective views for bounding the query could be discovered efficiently. The indexing structure can be adapted to the requirements of the tolerable error rate. Being an approximate solution, the exact results however are often returned. The experimental results show that, with a small tolerable error rate, the efficiency of the BPQ processing can be improved in 1-3 orders of magnitude by our proposed approximate solution over the exact solutions.

8. ACKNOWLEDGMENTS

This work is supported by NSFC under the grant No. 61003085 and the grant No. 61073019, and HGJ PROJECT 2010ZX01042-002-002-03.

9. REFERENCES

[1] B. Arai, G. Das, D. Gunopulos, and N. Koudas. Anytime measures for top-k algorithms. In *VLDB*, pages 914–925, 2007.

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

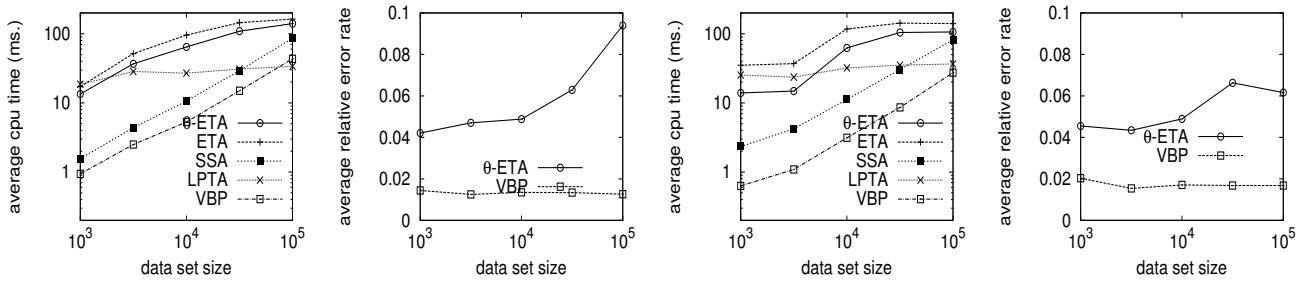
[3] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, pages 369–380, 2002.

[4] M. J. Carey and D. Kossmann. On saying "enough already!" in sql. In *SIGMOD Conference*, pages 219–230, 1997.

[5] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: indexing for linear optimization queries. *SIGMOD Rec.*, 29(2):391–402, 2000.

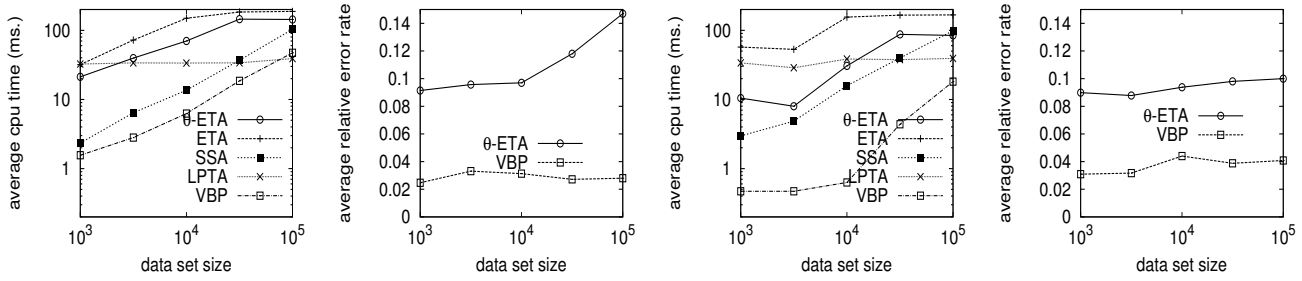
[6] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, pages 717–816, 2003.

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



(a) Efficiency, Household (b) Accuracy, Household (c) Efficiency, El Nino (d) Accuracy, El Nino

Figure 8: Performance comparison for queries of 4 dimensions



(a) Efficiency, Household (b) Accuracy, Household (c) Efficiency, El Nino (d) Accuracy, El Nino

Figure 9: Performance comparison for queries of 5 dimensions

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

[9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[10] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. *VLDB J.*, 16(1):5–28, 2007.

[11] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Selectivity estimators for multidimensional range queries over real attributes. *VLDB J.*, 14(2):137–154, 2005.

[12] V. Hristidis, and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *VLDB J.*, 13(1):49–70, 2004.

[13] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD Conference*, pages 259–270, 2001.

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

[15] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD Conference*, pages 635–646, 2006.

[16] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB*, pages 648–659, 2004.

[17] D. Xin, C. Chen, and J. Han. Towards robust indexing for ranked queries. In *VLDB*, pages 235–246, 2006.

[18] Z. Zhang, Y. Yang, R. Cai, D. Papadias, and A. K. H. Tung. Kernel-based skyline cardinality estimation. In *SIGMOD Conference*, pages 509–522, 2009.

[19] L. Zou and L. Chen. Dominant graph: An efficient indexing structure to answer top-k queries. In *ICDE*, pages 536–545, 2008.