Answering Top-k Queries with Multi-Dimensional Selections: The Ranking Cube Approach[*]

Dong Xin Jiawei Han Hong Cheng Xiaolei Li
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, 61801
{dongxin, hanj, hcheng3, xli10}@uiuc.edu

ABSTRACT

Observed in many real applications, a top-k query often consists of two components to reflect a user's preference: a *selection condition* and a *ranking function*. A user may not only propose *ad hoc* ranking functions, but also use different interesting subsets of the data. In many cases, a user may want to have a thorough study of the data by initiating a multi-dimensional analysis of the top-k query results. Previous work on top-k query processing mainly focuses on optimizing data access according to the ranking function only. The problem of efficient answering top-k queries with multi-dimensional selections has not been well addressed yet.

This paper proposes a new computational model, called *ranking cube*, for efficient answering top-k queries with multi-dimensional selections. We define a rank-aware measure for the cube, capturing our goal of responding to multi-dimensional ranking analysis. Based on the ranking cube, an efficient query algorithm is developed which progressively retrieves data blocks until the top-k results are found. The curse of dimensionality is a well-known challenge for the data cube and we cope with this difficulty by introducing a new technique of ranking fragments. Our experiments on Microsoft's SQL Server 2005 show that our proposed approaches have significant improvement over the previous methods.

1. INTRODUCTION

Ranking queries (or top-k queries) are commonplace in many real world applications, *e.g.*, searching Web database, k-nearest neighbor search with approximate matches and similarity queries in multimedia databases. A top-k query only returns the best k results according to a user-specified preference, which generally consists of two components: a *selection condition* and a *ranking function*. An example of a top-k query is shown as follows.

[*]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.

EXAMPLE 1. *Consider an online used car database R that maintains the following information for each car: type (e.g., sedan, convertible), maker (e.g., Ford, Hyundai), color (e.g., red, silver), transmission (e.g., auto, manual), price, milage, etc.. Two typical top-k queries over this database are:*

$$\begin{aligned} Q_1: \ & select\ top\ 10\ *\ from\ R \\ & where\ type = \text{“}sedan\text{”}\ and\ color = \text{“}red\text{”} \\ & order\ by\ price + milage\ asc \end{aligned}$$

$$\begin{aligned} Q_2: \ & select\ top\ 5\ *\ from\ R \\ & where\ maker = \text{“}ford\text{”}\ and\ type = \text{“}convertible\text{”} \\ & order\ by\ (price - 20k)^2 + (milage - 10k)^2\ asc \end{aligned}$$

Q_1 queries top-10 red sedans whose combined score over price and milage is minimized. Q_2 searches top-5 convertibles made by Ford, and the user expected price is $\$20k$ and expected milage is $10k$ miles.

The used car database may have other selection criteria for a car such as whether it has power window, air conditioner, sunroof, power steering, *etc.*. The number of attributes available for selection could be extremely large. A user may pick any subset of them and issue a top-k query using his/her preferred ranking function on the measure attributes (*e.g.*, price and milage). There are many other similar application scenarios, *e.g.*, the hotel search where the ranking functions are often constructed on the price and the distance to the interested area, and selection conditions can be imposed on the district of the hotel location, the star level, whether the hotel offers complimentary treats, internet access, *etc.*. Furthermore, in many cases, the user has his/her own criterion to rank the results and the ranking functions could be linear, quadratic or any other forms.

As shown in the above examples, different users may not only propose *ad hoc* ranking functions but also use different interesting subset of data. In fact, in many cases, users may want to have a thorough study of the data by taking a *multi-dimensional analysis* of the top-k query results. In the used car example given above, if a user is not satisfied by the top-5 results returned by Q_2, he/she may roll up on the maker dimension and check the top-5 results on all convertibles.

The above application scenarios propose a new challenging task for database system: *How to efficiently process top-k queries with multi-dimensional selection conditions and ad hoc ranking functions.* Given the sample queries in Example 1, current database systems will have to evaluate all the data records and output those top-k results which satisfy the selection conditions. Even if indices are built on each selection dimension, the database executer still needs to issue multiple random accesses on the data. This is quite expensive, especially when the database is large. Recent work on rank-aware data organization includes Onion [8], which builds convex hulls on data records according to their geometry relations and answers top-k queries by progressively retrieving data with levels; and PREFER [16], which creates ranked views and answers top-k queries by mapping query parameters to view parameters. Both approaches assume the ranking functions are linear, and hence have limitations to answer other common ranking functions. Moreover, their data organizations are not aware of the multi-dimensional selection conditions. A closely related study is the top-k selection queries proposed in [4], where the authors proposed to map a top-k selection query to a range query. The *soft* selection conditions in their queries are essentially the ranking functions for the k nearest neighbor search and our problem of answering top-k queries with *hard* selection conditions is not considered.

For multi-dimensional analysis, data cube [12] has been extensively studied. Materialization of a data cube is a way to pre-compute and store multi-dimensional aggregates so that online analytical processing can be performed efficiently. Traditional data cubes store the basic measures such as SUM, COUNT, AVERAGE, *etc.*, which are not able to answer complicated top-k queries. To the best of our knowledge, the problem of efficient processing top-k queries with multi-dimensional selection conditions is not well addressed yet.

In this paper, we propose a new computational model, called *ranking cube*, for efficient answering multi-dimensional top-k queries. We define a rank-aware measure for the cube, capturing our goal of responding to multi-dimensional ranking analysis. Using the ranking cube, we develop an efficient query algorithm. The curse of dimensionality is a well-known challenge for data cube and we cope with this difficulty by introducing a new technique of ranking fragments. More specifically, the contribution of this paper can be summarized as follows.

Rank-aware data cubing: We propose a new data cube structure which takes ranking measures into consideration and supports efficient evaluation of multi-dimensional selections and *ad hoc* ranking functions simultaneously. The "measure" in each cell is a list of tuple-IDs and its geometry-based partition facilitates efficient data accessing.

Query execution model: Based on the ranking cube, we develop an efficient query algorithm. The computational model recognizes both the progressive data accesses and the block-level data accesses.

Ranking fragments: To handle high-dimensional rank queries and overcome the curse-of-dimensionality challenge, we introduce a semi-materialization and semi-online computation model, where the space requirement for materialization grows linearly with the number dimensions.

The remainder of the paper is organized as follows. In Section 2, we formally present the problem formulation. In Section 3, we introduce the ranking cube structure, as well as the query execution algorithm based on the ranking cube. Section 4 describes the design of ranking fragments. Our performance study is presented in Section 5. We discuss the related work and the possible extensions in Section 6, and conclude our study in Section 7.

2. PROBLEM FORMULATION

Consider a relation R with categorical attributes $A_1, A_2, \ldots, A_S$ and real valued attributes $N_1, N_2, \ldots, N_R$. A top-k query specifies the selection conditions on a subset of categorical attributes and formulates a ranking function on a subset of real valued attributes. The result of a top-k query is an ordered set of k tuples that is ordered according to the given ranking function. A possible SQL-like notation for expressing top-k queries is as follows:

$$\begin{aligned}&select\ top\ k\ *\ from\ R\\ &where\ A'_1 = a_1\ and\ \ldots\ A'_i = a_i\\ &order\ by\ f(N'_1, \ldots, N'_j)\end{aligned}$$

Where $\{A'_1, A'_2, \ldots, A'_i\} \subseteq \{A_1, A_2, \ldots, A_S\}$ and $\{N'_1, N'_2, \ldots, N'_j\} \subseteq \{N_1, N_2, \ldots, N_R\}$. The results can be ordered by score ascending or descending order. Without losing generality, we assume the score ascending order is adopted in this paper.

We further notate A_i $(i = 1, \ldots, S)$ as *selection dimension* and N_i $(1, \ldots, R)$ as *ranking dimension*. In general, a real valued attribute can also be a selection dimension if it is discretized. A categorical attribute can also be a ranking dimension if the distance is defined on the values in the domain. In this paper, we assume the number of ranking dimensions is relatively small (*e.g.*, 2-4), while the number of selection dimensions could be rather large (*e.g.*, more than 10). This is a typical setting for many real applications. For example, both the used car database and the hotel database only have a limited number of ranking dimensions (*e.g.*, price, milage, distance). While the number of selection dimensions is much larger. Our proposed method can be naturally extended to cases where the number of ranking dimensions is also large (See Section 6).

For simplicity, we demonstrate our method using the *convex* ranking functions. Its extension to other functions will be discussed later in this paper. The formal definition of the convex function is presented in Definition 1.

DEFINITION 1. ***(Convex Function** [23]) A continuous function f is convex if for any two points x_1 and x_2 in its domain $[a, b]$, and any λ where $0 < \lambda < 1$:*

$$f(\lambda x_1 + (1 - \lambda)x_2) \leq \lambda f(x_1) + (1 - \lambda)f(x_2)$$

The convex functions already cover a broad class of commonly used functions. For example, all linear functions are convex. Note we made no assumption on the linear weights and they can be chosen either positive or negative. Hence the convex functions are more general to the commonly discussed linear *monotone* functions where the weights are restricted to be non-negative. Many distance measures are also convex functions. Suppose a top-k query looks for k tuples $t = (t_1, \ldots, t_r)$ which are the closest to the target value $v = (v_1, \ldots, v_r)$, both the ranking functions $f(t) = \sum_{i=1}^{r}(t_i - v_i)^2$ and $f(t) = \sum_{i=1}^{r} |t_i - v_i|$ are convex.

3. RANKING CUBE

In this section, we introduce the new computational model: *ranking cube*. We show the structure of the cube, the construction method and the techniques to answer top-k query using ranking cube. We first consider the case where the number of selection dimensions is small so that computing a full ranking cube over all selection dimensions is possible. The extension to larger number of selection dimensions can be handled by *ranking fragments*, which will be presented in next section.

3.1 Cube Structure

3.1.1 General Principles

Suppose a relation R has selection dimensions ($A_1, A_2, A_3, \ldots, A_S$) and ranking dimensions ($N_1, N_2, \ldots, N_R$). We build the ranking cube on the selection dimensions, and thus the multi-dimensional selection conditions can be naturally handled by the cube structure. To efficiently answer top-k queries with *ad hoc* ranking functions, the measure in each cell should have rank-aware properties.

A naïve solution is to put all the related tuples with their values on the ranking dimensions in each cell. This approach has two limitations: First, it is not space efficient because generally real values consume large spaces; and second, it is not rank-aware because the system does not know which tuple should be retrieved first w.r.t. an *ad hoc* ranking function.

To reduce the space requirement, we can only store the tuple IDs (*i.e.*, *tid*) in the cell. We propose two criteria to cope with the second limitation: the *geometry-based partition* and the *block-level data access*. The first one determines which tuples to be retrieved, and the second one defines how the tuples are retrieved. To respond to ranked query efficiently, we put the tuples that are geometrically close into the same block. During the query processing, the block which is the most promising to contain top answers will be retrieved first, and the remaining blocks will be retrieved progressively until the top-k answers are found.

3.1.2 Geometry Partition

Based on the above motivation, we create a new dimension B (*i.e.*, *block dimension*) on the base table. The new block dimension organizes all tuples in R into different rank-aware blocks, according to their values on ranking dimensions. To illustrate the concept, a small database, Table 1, is used as a running example. Let A_1 and A_2 be categorical attributes, and N_1 and N_2 be numerical attributes.

tid	A_1	A_2	N_1	N_2
1	1	1	0.05	0.05
2	1	2	0.65	0.70
3	1	1	0.05	0.25
4	1	1	0.35	0.15
...	...	...	...	...

Table 1: An Example Database

Suppose the block size is P. There are many ways to partition the data into multiple blocks such that (1) the expected number of tuples in each block is P, and (2) the tuples in the same block are geometrically close to each other. One possible way is the *equi-depth partition* [22] of each ranking dimension. The number of bins b for each dimension can be calculated by $b = (\frac{T}{P})^{\frac{1}{R}}$, where R is the number of ranking dimensions and T is the number of tuples in the database.

There are other partition strategies, *e.g.*, equi-width partition, multi-dimensional partition [20], *etc.*. For simplicity, we demonstrate our method using equi-depth partitioning. Our framework accepts other partitioning strategies and we will discuss this in section 6. Without loss of generality, we assume that the range of each ranking dimension is $[0, 1]$. We refer the partitioned blocks as *base blocks*, and the new *block dimension* B contains the *base block IDs* (simplified as *bid*) for each tuple. The original database can be decomposed into two sub-databases: the *selection database* which consists of the selection dimensions and the new dimension B, and the *base block table* which contains the ranking dimensions and the dimension B. The equi-depth partitioning also returns the meta information of the bin boundaries on each dimension. Such meta information will be used in query processing. Example 2 shows an equi-depth partitioning of the sample database.

EXAMPLE 2. *Suppose the data is partitioned into* 16 *blocks (Fig. 1). The original database is decomposed into two sub-databases as shown in Table 2. The bid is computed according to sequential orders (e.g., the four blocks on the first row are* b_1, b_2, b_3, b_4*; the four blocks on the second row are* b_5, b_6, b_7, b_8*; etc.). The selection dimension coupled with the dimension B will be used to compute ranking cube, and the ranking dimension table keeps the original real values. The meta information returned by the partitioning step is the bin boundaries:* $Bin_{N1} = [0, 0.4, 0.45, 0.8, 1]$ *and* $Bin_{N2} = [0, 0.2, 0.45, 0.9]$.

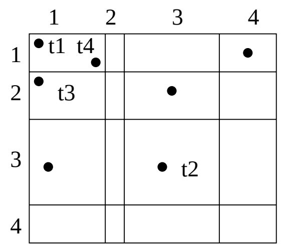

Figure 1: Equi-Depth Partitioning

tid	A_1	A_2	B
1	1	1	1
2	1	2	11
3	1	1	5
4	1	1	1
...	...	...	...

tid	B	N_1	N_2
1	1	0.05	0.05
2	11	0.65	0.70
3	5	0.05	0.25
4	1	0.35	0.15
...	...	...	...

Table 2: Table Decomposition

3.1.3 Rank-Aware Data Cubing

A cuboid in the ranking cube is named by the involved selection dimensions and ranking dimensions. For example, the cuboid $A_1A_2\_N_1N_2$ corresponds to selection dimensions A_1, A_2 and ranking dimensions N_1, N_2.

Our first proposal is to organize the *tid* list with respect to the different combinations of selection dimension and the dimension B (*e.g.*, $a_1^1a_2^1b_1$, where a_1^1, a_2^1 and b_1 are values on dimension A_1, A_2 and B). Since each *bid* represents a geometry region, the materialized ranking cube is able to quickly locate the cell which is the most promising for a given top-k query. Let us call the base blocks given by the equi-depth partitioning *logical blocks* and the disk blocks used to store the *tid* list *physical blocks.* Before the multi-dimensional data cubing, each logical block corresponds to one physical block. However, with the multi-dimensional data cubing, the tuples in each logical base block (*e.g.*, b_1) are distributed into different cells (*e.g.*, $a_1^1a_2^1b_1$, $a_1^2a_2^2b_1$, *etc.*), and thus the number of tuples in each cell is much smaller than the physical block size. In order to leverage the advantage provided by block-level data access on disk, we introduce the *pseudo block* as follows. The base block size is scaled in each cuboid such that the expected number of tuples in each cell occupies a physical block. Let the cardinality of the selection dimensions of a cuboid $A_1A_2\dots A_S\_N_1N_2\dots N_R$ be $c_1, c_2, \dots, c_s$, the expected number of tuples in each cell is $n = \frac{P}{(\prod_{j=1}^{s} c_j)}$, where P is the block size. The *scale factor* can be computed by $sf = \lfloor (\frac{T}{n})^{\frac{1}{s}} \rfloor = \lfloor (\prod_{j=1}^{s} c_j)^{\frac{1}{s}} \rfloor$. The pseudo block is created by merging every other sf bins on each dimension. We assign the *pseudo block ID* (or, *pid*) to the dimension B for each tuple, and the *bid* value is stored together with *tid* in the cell. A pseudo block partitioning is demonstrated in Example 3.

EXAMPLE 3. *In Table 2, let the cardinalities of A_1 and A_2 be 2, and there will be 4 pseudo blocks (Fig. 2). The solid lines are the partitions for pseudo blocks and the dashed lines are the original partitions for base blocks. The new cuboid is shown in Table 3.*

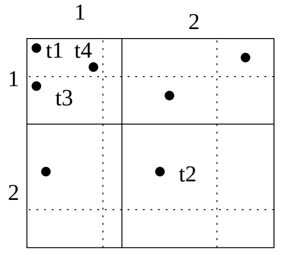

Figure 2: Pseudo Block Partitioning

A_1	A_2	pid	tid (bid) List
1	1	1	1(1), 3(5),4(1)
1	2	4	3(11)
...	...	...	...

Table 3: $A_1A_2\_N_1N_2$ Cuboid After Pseudo Blocking

Given a database with S selection dimensions and R ranking dimensions, a ranking cube consists of a triple $\langle T, C, M\rangle$, where T is the base block table on R ranking dimensions; C is the set $2^S - 1$ different ranking cuboids according to the combination of S selection dimension; and M is the set of meta information which includes the bin boundaries for each ranking dimension and the scale factors for each cuboid.

3.2 Online Query Computation

Here we discuss how to answer top-k queries using ranking cube. We first define the data access methods, then present the query processing algorithm. Finally, we demonstrate the algorithm by a running example.

3.2.1 Data Access Methods

We assume the ranking cube is stored on disk, and we only access data at the block-level. The two data access methods are: *get pseudo block* and *get base block.* The first one accesses the ranking cube. It accepts a cell identifier in a ranking cube, (*e.g.*, $A_1 = a_1, A_2 = a_2, pid = p_1$), and returns a list of *bid* and *tid* in the cell; and the second one accesses the base block table. It accepts a *bid* and returns a list of *tid* and their real values of the ranking dimensions.

Here we briefly explain why the combination of both data access methods may benefit the query processing. First, if the *get pseudo block* were the only available method, the query processing algorithm would be able to locate some promising *tid*s. However, it might need to issue multiple random accesses to retrieve the values of those *tid*s. The *get base block* method can reduce the number of random access, especially when the cardinalities are low. Second, if the *get base block* were the only available method, the algorithm might have wasted some I/O since some base blocks may not appear with the corresponding selection dimensions. The *get pseudo block* method can guide the system to access the right base blocks, especially when the cardinalities are high. Hence, the combination of these two data access methods provides a fairly robust mechanism.

3.2.2 Query Algorithm

Our query processing strategy consists of the following four steps: *pre-process, search, retrieve* and *evaluate.*

Pre-process: The query algorithm first determines the cuboid C and base block table T by the selection conditions and ranking functions.

Search: This step finds the next candidate base block for data retrieving. A *tid* may be retrieved twice, first from C and second from T. We say a tuple is *seen* if its real values are retrieved from T and the score w.r.t. the ranking function is evaluated. Otherwise, it is *unseen.* The algorithm maintains a list of scores S for the seen tuples and let the k^{th} best score in the list be S_k.

For each base block *bid*, we define $f(bid)$ as the best score over the whole region covered by the block. Given a ranking function, the algorithm computes the *bid* whose $f(bid)$ is minimum among all the remaining base blocks (A base block is not included for further computation after it is retrieved and evaluated). Let this score be S_{unseen} and the corresponding block be the *candidate block*. If $S_k \leq S_{unseen}$, the top-k results are found and the algorithm halts.

The key problem in this step is to find the candidate block. At the very beginning, the algorithm calculates the minimal value of the ranking function f. Since we assume f is convex, the minimal value can be found efficiently. For example, if f is linear, the minimum value is among the extreme points of whole region; if f is quadratic, the minimum value can be found by taking the derivative of f. The first candidate block corresponds to the block which contains the minimal value point. To search for the following candidate blocks, the result of Lemma 1 can be used.

LEMMA 1. *Assume the set of examined blocks is E. Let $H = \{b | neighbor(b, c) = true, \exists c \in E\}$, where $neighbor(b, c)$ returns true if blocks b and c are neighboring blocks. If the ranking function is convex, then the next best base block is $bid^* = \arg\min_{bid \in H} f(bid)$, where $f(bid) = min_{p \in bid} f(p)$.*

Based on Lemma 1, we maintain a list H, which contains the neighboring blocks of the previous candidate blocks. Initially, it only contains the first candidate block found by the minimum value of f. At each round, the algorithm picks the first block in H as the next candidate block and removes it from H. At the same time, the algorithm retrieves all the neighboring blocks of this candidate block and inserts them into H. Since each block can be neighboring with multiple blocks, we maintain a hash-table of inserted blocks so that each block will only be inserted once. The blocks in H are resorted by $f(bid)$ and the first one has the best score.

Retrieve: Given the *bid* of the candidate block computed in the search step, the algorithm retrieves a list of *tid*'s from the cuboid C. It first maps the *bid* to a *pid* and then uses *get pseudo block* method to get the whole pseudo block identified by *pid*. Since the mapping between *bid* and *pid* is many-to-one, it is possible that a *pid* block has already been retrieved in answering another *bid* request. To avoid multiple retrieving on the same pseudo block, we buffered the *bid* and *tid* lists retrieved so far. If a *bid* request maps to a previously retrieved *pid*, we directly return the results without accessing the cuboid C.

Evaluate: Given the *bid* computed in the search step, if the set of *tid*'s returned by the retrieve step is not empty, the algorithm uses the *get base block* method to retrieve the real values of those tuples. The real values are used to compute the exact score w.r.t. the ranking function f. The scores are further merged into the score list S maintained by the search step.

If the original query consists of other projection dimensions which are not in either the selection nor the ranking dimensions, we can further retrieve the data tuples from the original relation using the top-k *tid*s.

3.2.3 A Demonstrative Example

Using the database in Table 1, we demonstrate each step by a running example:

$$\begin{aligned}&select\ top\ 2\ *\ from\ R\\&where\ A_1 = 1\ and\ A_2 = 1\\&sort\ by\ N_1 + N_2\end{aligned}$$

The algorithm first determines that cuboid $C = A_1A_2\_N_1N_2$ can be used to answer the query. The related meta information is shown in Table 4.

Meta Info.	Value
Bin Boundaries of N_1	[0,0.4,0.45,0.8,1]
Bin Boundaries of N_2	[0,0.2,0.45,0.9,1]
scale factor of C	2

Table 4: Meta Information for answering query

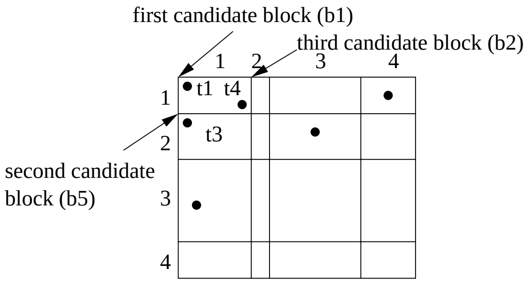
first candidate block (b1)
third candidate block (b2)
second candidate block (b5)
1
2
3
4
1
2
3
4
t1
t4
t3

Figure 3: Processing top-2 on example query

Suppose the range of each ranking dimension is $[0, 1]$. The minimal value of the function $f = N_1 + N_2$ is 0 (by $N_1 = 0$ and $N_2 = 0$). The algorithm locates the first base block as b_1 (as shown in Fig. 3), and asks cuboid C to return the *tid* list of this block. The cuboid C maps b_1 to p_1 by the scale factor 2 (as shown in Fig. 2). Then C issues the *get pseudo block* method and retrieves the following contents: $\{t_1(b_1), t_4(b_1), t_3(b_5)\}$. t_1 and t_4 are returned as the results of the b_1 query and $t_3(b_5)$ is buffered for future queries. The algorithm then issues the *get base block* method to get the real values of tuples in b_1 and verifies that the exact score of t_1 is 0.1 and that of t_4 is 0.5. To test the stop condition, the algorithm computes the neighboring blocks of b_1. In this example, the neighboring blocks are b_2 and b_5, and thus $H = \{b_2, b_5\}$. Using the meta information on bin boundaries, we can compute the best scores of the neighboring blocks w.r.t. to the ranking function $f = N_1 + N_2$. The base block b_2 has the best score 0.4 and the base block b_5 has the best score 0.2. Both correspond to the left upper points (as shown in Fig. 3). Hence, $S_{unseen} = f(b_5) = 0.2$. At this stage, the list of S and the list of H are shown in Table 5.

List	Scores
S list	$f(t_1) = 0.1$, $f(t_4) = 0.5$, $S_2 = 0.5$
H list	$f(b_2) = 0.4$, $f(b_5) = 0.2$, $S_{unseen} = 0.2$

Table 5: List Values at Stage 1

Since the current k^{th} score is $S_2 = 0.5 > 0.2 = S_{unseen}$, the stop condition is not met. The algorithm continues to pick b_5 as the next candidate block, and inserts its neighboring blocks b_6 and b_9 into H. Again, the cuboid C is asked to return the *tid* list with b_5, which is mapped to p_1. This time, the results are buffered and $\{t_3(b_5)\}$ is directly returned. The algorithm further retrieves real values for b_5 and verifies that the exact score of t_3 is 0.3. After updating the score list S, we have $S_2 = 0.3$. The list H contains blocks b_2, b_6 and b_9, and the scores associated with them are $f(b_2) = 0.4$, $f(b_6) = 0.6$, and $f(b_9) = 0.45$, respectively. Thus, $S_{unseen} = 0.4$. At this stage, the list of S and the list of H are shown in Table 6. Since $S_2 = 0.3 <= S_{unseen}$, the stop condition is satisfied. The algorithm returns t_1 and t_3 as top-2 results.

List	Scores
S list	$f(t_1) = 0.1$, $f(t_3) = 0.3$, $f(t_5) = 0.5$, $S_2 = 0.3$
H list	$f(b_2) = 0.4$, $f(b_9) = 0.45$, $f(b_6) = 0.6$, $S_{unseen} = 0.4$

Table 6: List Values at Stage 2

4. RANKING FRAGMENTS

When the number of selection dimensions is large, a full materialization of the ranking cube is too space expensive. Instead, we adopt a *semi-online computation model with semi-materialization.*

Before delving deeper into the *semi-online computation*, we claim the following observation about ranked query in high-dimensional space. Although a database may contain many selection dimensions, most queries are performed only on a small number of dimensions at a time. In other words, a real life ranked query is likely to ignore many selection dimensions. Stemming from the above observation, we partition the dimensions into different groups called *fragments*. The database is projected onto each fragment, and ranking cubes are fully materialized for each fragment. With the semi-materialized fragments, one can dynamically assemble and compute any ranking cuboid cells of the original database online. In the rest of this section, we discuss the two components of our computation model: *semi-materialization* and *semi-online computation.*

4.1 Materializing Fragments

Here we show a general grouping framework and its storage size analysis. There are other criteria which can be exploited to group selection dimensions for efficient query processing, and we will address these issues in Section 6. Suppose the database has S selection dimensions and the size of the fragment is F (*i.e.*, the number of selection dimensions in the fragment), we evenly partition the selection dimensions into $\frac{S}{F}$ disjoint sets. Each fragment will combine with the ranking dimensions to construct a ranking cube. An example of fragment grouping is shown in Example 4.

Example 4. *Suppose a relation has* 4 *selection dimensions* $A_1, \ldots, A_4$ *and two ranking dimensions* N_1, N_2. *We evenly group the selection dimensions into two fragments* (A_1, A_2) *and* (A_3, A_4) *and the ranking fragments are:* (A_1, A_2, N_1, N_2) *and* (A_3, A_4, N_1, N_2).

We estimate the space consumption for the ranking fragments. Given a relation with S selection dimensions, R ranking dimensions, and T tuples, let the fragment size be F. There will be total $\frac{S}{F}$ fragments, while each fragment has $O(2^F - 1)$ cuboids. Each cell in a cuboid stores the *bid* and *tid* lists. Since *tid*'s are exclusively stored in different cells, the size of each cuboid is $2T$. The base block table has $R + 2$ dimensions (*i.e.*, including the *bid* and *tid*). The overall size of base block tables is $O((R + 2)T)$. The meta information for each fragment can be neglected, comparing with the size of the cuboids and the base block tables. The above estimation is summarized as the following lemma.

Lemma 2. *Given a database with S selection dimensions, R ranking dimensions and T tuples, the amount of disk space needed to store the ranking fragments with size F is* $\mathcal{O}(2\frac{S}{F}T(2^F - 1) + (R + 2)T)$.

Based on Lemma 2, for a database with 12 selection dimensions and 2 ranking dimensions, using fragment size $F = 2$, the total amount of space requirement is on the order of $2T(\frac{12}{2})(2^2 - 1) + (2 + 2)T = 40T$. Suppose *tid*, *bid* and each dimension in the database take same unit storage space, this is around 3 times the size of the original database. One can verify that given a fixed fragment size, the space consumption by the ranking fragments grows linearly with the number of selection dimensions.

4.2 Answering Query by Fragments

Given the semi-materialized ranking fragments, one can answer a ranked query on the original data space. We say a cuboid *covers* a query if all the dimensions involved in the query appear on the materialized cuboid. In this case, the query can be directly answered using the query algorithm described in Section 3.2. Otherwise, the query is answered by a set of cuboids which, as a whole, cover the query.

4.2.1 Determining Covering Cuboids

The covering cuboids set can be determined by a *minmax* criterion. More specifically, let the set of selection dimensions contained in a cuboid C be $Dim(C)$. Suppose the set of selection dimensions in the query is Q. To determine which cuboid to be used to answer the query, we first find all cuboids C such that there is no other cuboid C' satisfying $Dim(C) \subsetneq Dim(C') \subseteq Q$ (maximum step). Let the set of cuboids returned by the above step is MD, the second step searches for a minimum subset $MS \subseteq MD$ such that $Q = \cup_{C \in MS} Dim(C)$. An example of determining cuboids is shown as below.

Example 5. *Suppose the materialized fragments are* (A_1, A_2, N_1, N_2) *and* (A_3, A_4, N_1, N_2). *The query consists of the selection dimensions* (A_1, A_4) *and ranking dimensions* (N_1, N_2). *We first locate the set of candidate cuboids* $MD = \{A_1\_N_1N_2, A_4\_N_1N_2\}$, *and this is also the minimum covering subset* MS.

4.2.2 Computing Cuboid Cells Online

We discuss how to answer queries by a set of ranking fragments. The general idea is to online compute the cuboid covers the query. Instead of computing the whole cuboid, we only compute the cells which is required to answer the query. The online computation is made efficient by set intersection operation on the *tid* lists.

Suppose the query has the selection dimensions (A_1, A_4). The fragments demonstrated in Example 4 are used and a covering set of two cuboids $A_1\_N_1N_2$ and $A_4\_N_1N_2$ is selected. Our goal is to online compute the required cells of cuboid $A_1A_4\_N_1N_2$. Based on the query algorithm presented in Section 3.2, we only need to make a small change at *retrieve* step. Instead of issuing the *get pseudo block* method to cuboid $A_1A_4\_N_1N_2$, which was not materialized, we issue the *get pseudo block* method to cuboid $A_1\_N_1N_2$ and $A_4\_N_1N_2$. The *tid* lists returned by both cuboids will be *intersected* as the answer. All the other steps in the query algorithm (Section 3.2) remain the same. The *merge and intersect* operation on the tif lists can be generalized to more than two ranking fragments.

5. PERFORMANCE STUDY

This section reports the experimental results. We compare the query performance of ranking cube and ranking fragments with two other alternatives: the *baseline* solution by Microsoft SQL-Server and the *rank mapping* approach proposed by [4]. We first discuss the experimental settings, and then show the results on low dimensional data (with ranking cube) and high dimensional data (with ranking fragments).

5.1 Experimental Setting

We defines the data sets, the experimental configurations for all three methods and the evaluation metric.

5.1.1 Data Sets

We use both synthetic and real data sets for the experiments. The real data set we consider is the *Forest CoverType* data set obtained from the UCI machine learning repository web-site (www.ics.uci.edu/∼mlearn). This data set contains $581,012$ data points with 54 attributes, including 10 quantitative variables, 4 binary wilderness areas and 40 binary soil type variables. We select 3 quantitative attributes (with cardinalities $1,989$, $5,787$ and $5,827$) as ranking dimensions, and other 12 attributes (with cardinalities 255, 207, 185, 67, 7, 2, 2, 2, 2, 2, 2, 2) as selection dimensions. To achieve a reasonable size of the data, we further duplicate the original data set 5 times and the data set has $3,486,072$ tuples. We also generate a number of synthetic data sets for our experiments. The parameters and default values are summarized in Table 7.

Parameter	Default Value
S: Number of selection dimensions	3 (Cube)
	12 (Fragments)
R: Number of ranking dimensions	2
T: Number of Tuples	$3M$
C: Cardinality	20

Table 7: Parameters for Synthetic Data Sets

5.1.2 Experimental Configurations

In the experiments, we compare our proposed approach against the baseline solution provided by commercial database and the rank mapping technique discussed in [4]. All the experiments are conducted over Microsoft SQL Server 2005 on a 3GHz Pentium IV PC with 1.5GBytes of RAM. Specifically, we use the following techniques for answering top-k queries.

Baseline Approach: We load all experimental data sets into SQL Server 2005. A non-clustered index is built on each selection dimension. The baseline performance is measured by simply issuing the following SQL statement to the SQL Server:

$$\begin{aligned}&select\ top\ k\ *\ from\ D\\&where\ A_1 = a_1\ and\ \ldots\ A_i = a_i\\&order\ by\ f(N_1, \ldots, N_j)\end{aligned}$$

where A_i belong to selection dimensions and N_i belong to ranking dimension.

Rank Mapping Approach: In [4], the authors proposed to map a top-k selection query to a range query. Their problem definition is slightly different from ours since the top-k queries in [4] do not have *hard* selection condition. However, the idea of mapping a ranking function to a range query can also be applied in our problem. We refer their method as *rank mapping*. An example of applying rank mapping in our problem is as follows:

$$\begin{aligned}Top\text{-}k\ Query:\ &select\ top\ k\ *\ from\ D\\&where\ A_1 = a_1\ and\ \ldots\ A_i = a_i\\&order\ by\ N_1 + 2N_2\end{aligned}$$

$$\begin{aligned}Range\ Query:\ &select\ top\ k\ *\ from\ D\\&where\ A_1 = a_1\ and\ \ldots\ A_i = a_i\ and\\&N_1 \le \overline{n_1}\ and\ N_2 \le \overline{n_2}\\&order\ by\ N_1 + 2N_2\end{aligned}$$

The performance of this approach relies on two aspects: (1) how the bound values $\overline{n_1}$ and $\overline{n_2}$ are determined; and (2) how the index in the database is configured to efficiently answer the multi-dimensional range query.

The original proposal for the first issue is to use a workload adaptive mapping strategy to provide the selectivity estimation. Since the workload information is not available to us in our experiment, we make an extremely conservative comparison by feeding the rank mapping approach the *optimal* bound values. For the sample query given above, if the final k^{th} tuple in the result is evaluated as 100 by the ranking function, we assign $\overline{n_1}$ as 100 and assign $\overline{n_2}$ as 50. This is the best estimation that any mapping strategy can provide.

For the second issue, the original proposal is to build a multi-dimensional index on all participating attributes. We will continue to use it when we test performance on ranking cube, where the number of involved dimensions is comparatively low. The dimension order in the index is first the selection dimensions and then the ranking dimensions. For our ranking fragments experiment, a single multi-dimensional index is not practical since the number of dimension is quite high. Instead, we build several partial multi-dimensional indices and each of them corresponds to one ranking fragment.

Ranking Cube (Fragments): For a fair comparison, we load the ranking cube (fragments) into SQL Server. To simulate the block-level access, we build a clustered index on selection dimensions A_i and the pseudo block ID (*pid*) for each cuboid; and a clustered index on base block ID (*bid*) for the base block table. We implement our query algorithms using Visual.net *c#*. The ranking cube (fragments) have two parameters: the base block size B and the fragment size F. By default, we set B as 300 and F as 2. We will conduct experiments to examine the query performance with respect to these two parameters.

5.1.3 Evaluation Metric

We use *execution time* to evaluate the techniques presented above. For each experiment, we report the average running time for executing a set of 20 randomly issued queries. Without loss of generality, we use linear ranking functions in our evaluation. One criterion to measure the query difficulties of the linear ranking functions is the query skewness, which is defined as follows. For a linear ranking function $\alpha_1 N_1 + \alpha_2 N_2 + \ldots + \alpha_r N_r$, let $\underline{\alpha} = \min_{i=1}^{r} \alpha_i$ and $\overline{\alpha} = \max_{i=1}^{r} \alpha_i$, the query skewness is defined as $u = \overline{\alpha}/\underline{\alpha}$.

The parameters and their default values for queries are shown in Table 8.

Parameter	Default Value
s: Number of selection conditions	2
r: Number of dimensions involved in ranking function	2
k: Number of top results requested	10
u: Query Skewness	1

Table 8: Parameters for Queries

5.2 Experiments on Ranking Cube

This section presents experimental results for the top-k query processing using ranking cube. We use synthetic data set in this set of experiments. To study the query performance with respect to different criteria, we vary the value of k (the number of top results requested), u (query skewness), T (the number of tuples in the database), C (the cardinalities of each dimension), s (the number of selection conditions), r (the number of dimensions involved in the ranking function) and B (the base block size). Another important measure is the space requirement, and we will report the result in the next subsection. All the parameters in the data sets and queries use the default values (if not explicitly specified).

Top-k Query: Figure 4 reports the execution time as a function of k (*i.e.*, the number of tuples requested) on the default synthetic data. Our methods is much more efficient than the previous approaches. As expected, the baseline approach is not sensitive to the value of k since it retrieves all the tuples. The execution time of the rank mapping approach increases slightly. This is because we assign the optimal bound values for range queries and those bound values increase slightly. Our method progressively retrieves data blocks, and thus a larger k value asks for more data accesses. When $k = 10$, the ranking cube is 4 times faster than the rank mapping approach and 10 times faster than the baseline approach.

Query Skewness: In this experiment, we measure the performance by varying query skewness. The goal of this experiment is to test the robustness of the base block partitioning. A skewed query may require the system to retrieve more data since the top results may be distributed on more base blocks. We vary the value of u and the results are reported in Figure 5. The execution time of our method increases slightly with u. However, it still performs much better comparing with other alternatives.

Number of Dimensions in Ranking Function: Here we continue to test the query performance with respect to the ranking function. We generate a synthetic data with 3 selection dimensions and 4 ranking dimensions, and vary the value of r (the number of dimensions involved in the ranking function) from 2 to 4. The results are shown in Figure 6. We observe that the execution time of our method slightly increases when the number of dimensions decreases. This is because our base block table is constructed on all 4 ranking dimensions. A 2 dimensional query means the blocks need to be projected onto the lower dimensions. Hence more block accesses are required. The baseline approach is not sensitive to r. The rank mapping approach performs worse because the bound estimation, although it is optimal, is much looser in higher dimensions.

Database Size: To analysis the query performance with respect to the data size, we change the number of tuples in the synthetic data from $1M$ to $10M$ (Figure 7). The baseline performs worse on larger data set since the selection conditions return more qualified tuples and the database needs to do more random accesses. Although the rank mapping approach is fed by the optimal bound values, it performs worse with larger data size. This may be caused by the reason that query execution time is sensitive to the dimensions involved in the query. If the involved dimensions exactly follow the order on which the multi-dimensional index was built, it can be answered extremely fast. Otherwise, there will be more random access and the execution time is also affected by the data size. On the other hand, the ranking cube approach has stable performance, regardless of the data size. This indicates that our proposed approach is especially attractive for larger data set.

Cardinality: We generate a set of synthetic data by varying the cardinality of each selection dimension from 10 to 100. The results are shown in Figure 8. Basically, increasing cardinality favors the baseline approach since the number of tuples filtered by the selection conditions decreases significantly. There is no clear trend of the rank mapping approach in this experiment, mainly because we assign the optimal bound value for the transformed range query and the number of tuples satisfying the range query is thus not sensitive to the cardinalities. For our method, the scale factor (see Section 3) of the pseudo block increases with the cardinality and the tuples are more sparse in each pseudo block. As the result, it invokes more *get base block* methods to verify the tuples and the execution time slightly increases. When C is large enough (*i.e.*, 100), we observe the execution time decreases again. This is because the number of tuples retained by the selection conditions is quite small and many base blocks are not retrieved since they are found to be empty during the *get pseudo block* step. This is consistent with our analysis that the combination of two block access methods is robust (see Section 3.2.1).

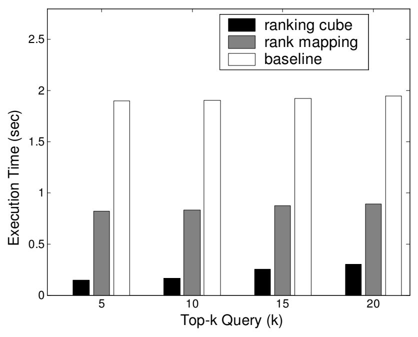

Figure 4: Query Execution Time w.r.t. k

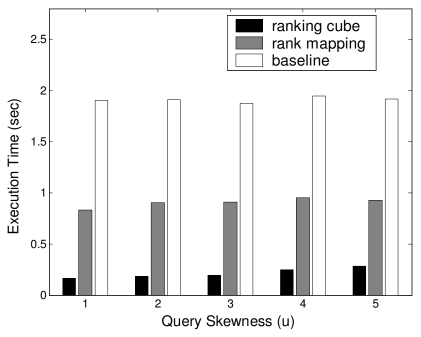

Figure 5: Query Execution Time w.r.t. u

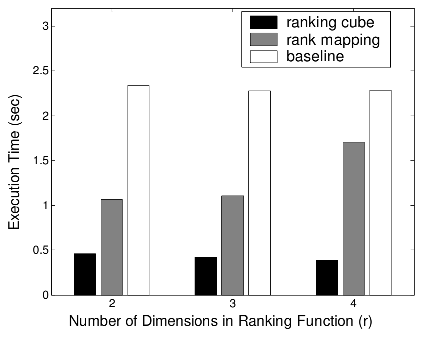

Figure 6: Query Execution Times w.r.t. r

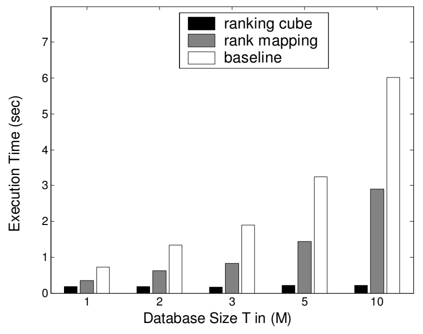

Figure 7: Query Execution Time w.r.t. T

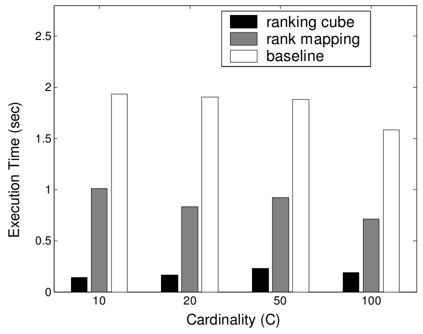

Figure 8: Query Execution Time w.r.t. C

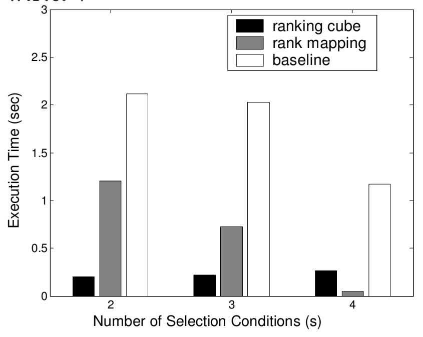

Figure 9: Query Execution Time w.r.t. s

Number of Selection Conditions: Here we vary the number of selection conditions to test the trade-offs between the ranking and selection. The results are shown in Figure 9. Generally, involving more selection conditions results in fewer qualified tuples, and consequently, the baseline approach improves its query performance. The execution time of rank mapping decreases because the multi-dimensional index has better utilization. In the experiment, we use a synthetic data with 4 selection dimensions. When the dimensions involved in the query are the same as those in the multi-dimensional index (*i.e.*, $s = 4$ in Figure 9), the range query can be answered very fast. The execution time of ranking cube slightly increases with the number of selection conditions, and overall its performance is not sensitive to the number of selection conditions, for the same reason given in the previous experiment. We also observe that with 4 selection conditions, the number of qualified tuples is 22. Ranking is even not necessary in this case.

Block Size: The final experiment with ranking cube is to test the sensitivity with respect to the block size B. We measure the block size by the expected number of tuples contained by a block. The results are shown in Figure 10. We observe that the execution time is within 10% between each other cases, and the performance not sensitive to the value of B (from 100 to 1000). In our experiments, we use 300 as the default value for B.

5.3 Experiments on Ranking Fragments

In this subsection, we present experimental results on ranking fragments. We first examine the cost of storing the fragments, and then test the query execution time. Both synthetic and real data in our experiments have 12 selection dimensions. We increase the default number of selection conditions to 3.

Space Consumption: The first experiment is to examine the amount of space needed to store the ranking fragments. Specifically, how it scales as the dimensionality grows. We use a synthetic data with 3-12 selection dimensions and build ranking fragments with $F = 2$. Figure 11 shows the total space consumption in SQL server. The space usage includes both the data and the indices. We compare the total space usage with baseline (BL) and rank mapping (RM) approaches. The baseline approach builds a non-clustered index on each selection dimension and the rank mapping approach builds a multi-dimensional index for each ranking fragment. Although neither of them generates new tables, the indices used by them are much larger than the the base table. We also observe that the clustered indices built by the ranking fragments (RF) occupy small space (roughly 1% of the total space). As shown in the figure, the space usage of all three methods grow linearly with the number of dimensions. The space used by ranking fragments is only 2-2.5 times of that of the other two alternatives. It is a fairly acceptable cost paid for materialization since the online query processing becomes much more efficient. Comparing with other data cube proposals, the space requirement by the ranking fragments is more practical. Furthermore, since we store ranking fragments in relational database, a large portion of the space is used to store the cell identifiers. We believe that the space requirement can be further reduced if we store the data out of the relational database. We discuss more compression opportunities in Section 6.

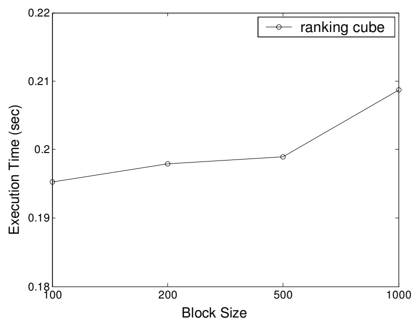

Figure 10: **Query Execution Time w.r.t. Block Size**

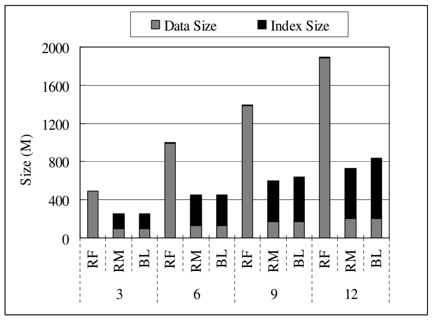

Figure 11: Space Usage w.r.t. Number of Selection Dimensions

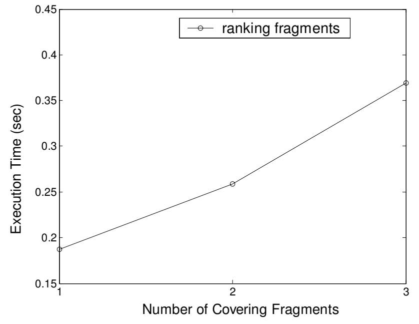

Figure 12: Query Execution Time w.r.t. Number of Covering Fragments

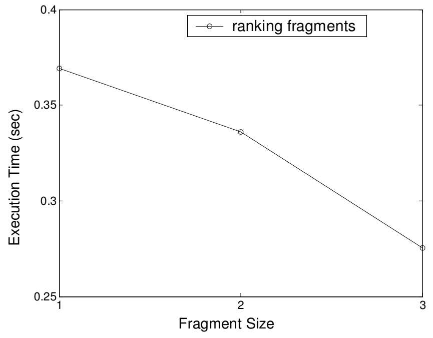

Figure 13: Query Execution Time w.r.t. Fragment Size

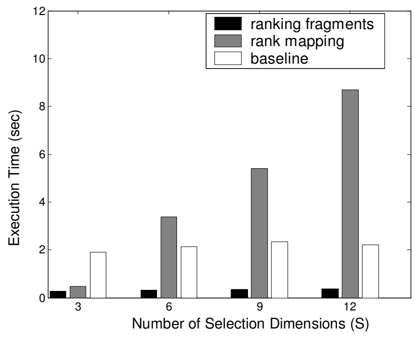

Figure 14: Query Execution Time w.r.t. S

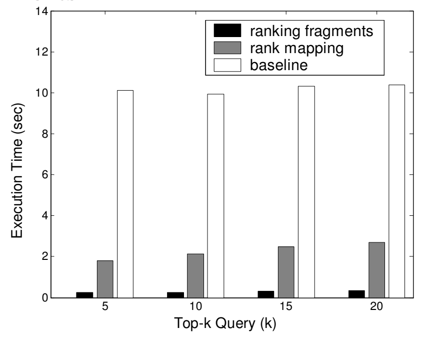

Figure 15: Query Execution Time on Real Data

Number of Covering Fragments: Since the ranking fragments do not guarantee to cover a query by a single fragment, we test the query performance with respect to the number of covering fragments. We generate 3 top-k queries, each involving 3 selection dimensions, and intentionally let them be covered by one, two and three fragments. We use the same data set as described above, and the execution time is shown in Figure 12. The execution time increases with the number of covering fragments. Typically, the execution time with 2 (3) covering fragments is roughly 1.4 (2) times of that with one covering fragment. We observe that even with 3 covering fragments, our method is still around 4 times faster than the baseline and 2 times faster than the rank mapping approach (See Figure 4).

Fragment Size: Here we test the query performance with respect to the fragment size. A larger fragment size will have better coverage of the queries, however, it requires more space usage as well. Generally, fragment size larger than 3 is too space consuming, and we test the performance on fragment sizes 1, 2, and 3. We continue to use the same synthetic data set described above and the queries are generated with 3 selection conditions. The results are shown in Figure 13. We observe that the larger fragment size provides better query performance.

Number of Dimensions: Here we fixed the fragment size as 2 and vary the number of selection dimensions from 3 to 12. We compare the query execution time with the baseline and the rank mapping approaches. The results are shown in Figure 14. We have the following observations. First, the baseline approach is not sensitive to the number of dimensions. Second, the rank mapping approach becomes worse when dimension increases. This is because in high dimensional data, the multi-dimensional index in each fragment has low probability to cover a query. In many cases, the query only accesses one dimension in a multi-dimensional index and this is quite expensive. Finally, the time used by ranking fragments increases slightly from 3 to 9 dimensions. For queries with three selection conditions, we expect that the execution time will keep stable with higher dimensions. This is because query execution in the worst case (*i.e.*, the query is covered by three different fragments) is still quite efficient (see Figure 12).

Real Data: Besides synthetic data, we also tested our proposed methods on the real-world data set: Forest CoverType (see Section 5.1). We evenly partition the 12 selection dimension into 4 groups (*i.e.*, fragment size is 3). The queries has 3 selection conditions and the ranking function spans on all three ranking dimensions. The query execution time with respect to k is shown in Figure 15. Different from the result in last experiment, here we observe that the rank mapping approach is more efficient than the baseline approach. This is because in this real data set, many dimensions have cardinality 2. As a result, the baseline approach needs to access more tuples. The low cardinalities also enable the rank mapping approach to efficiently access the multi-dimensional indices. Comparing with these two alternatives, our proposed method consistently performs the best on both the synthetic and read data sets.

6. DISCUSSION

In this section, we discuss the related work and possible extensions if our proposed approach.

6.1 Related Work

Top-k query processing has been studied in both the middleware scenario [13, 14, 7] and in the relational database setting [5, 4, 9, 17, 18, 16]. These studies mainly discuss the configurations where only ranking dimensions are involved, the problem of top-k queries with multi-dimensional selections is not well addressed. The closest related work may be the top-k selection query problem studied in [4], which maps a top-k nearest neighbor (*e.g.*, soft-selection) query to a range query. The problem is essentially a top-k query using a distance measure as ranking function, and there is no multi-dimensional selections imposed on queries.

We organize tuples into different blocks based on their geometry layout information. Previous work on exploiting data distributions for efficient query processing includes [8, 1, 15]. The ranking function covered in those studies are linear. We have demonstrated our method with convex functions and the extension to *ad hoc* ranking functions is fairly straightforward (see Section 6.2.1).

Data Cube has been playing an essential role in the implementation of fast OLAP operations [12]. Materialization of a data cube is a way to pre-compute and store multi-dimensional aggregates so that multi-dimensional analysis can be performed on the fly. The pre-computed measures in the cube are generally simple statistics (*e.g.*, SUM, COUNT, AVERAGE). Some recent proposals introduces more complex measures for data cube such as linear regression model [11] and classification model [10]. To the best of our knowledge, this is the first piece of work that provides multi-dimensional ranking analysis using data cube.

The *tid* list stored in the ranking cube is similar to the ideas of inverted index as termed in the information retrieval and value-list index as termed in databases. In [3], the authors investigated the usage of low dimensional data structures for indexing a high dimensional space. Their data structures and algorithms were only designed to index data points, with measure aggregation. The model of multi-dimensional inverted index and measure aggregation is studied by [19]. The major difference of our approach and all these studies is our *bid* list is rank-aware and it supports progressive retrieving for efficient processing of top-k queries. Moreover, the contents in the ranking cube could be extended to contain rich information such as the max (min) number of tuples in each block, the mean values of tuples in each block, *etc.*. Extending the ranking cube framework to efficiently support more complicated ranked queries is an interesting future research direction.

6.2 Extensions

6.2.1 Ad Hoc Ranking Functions

Here we discuss the extension to *ad hoc* ranking functions. The basic idea is to decompose the whole domain of the function variables into multiple sub-domains so that in each sub-domain, the function has convex property. The decomposition relies on the scientific computing techniques and is out of the scope of this paper. After the convex sub-domains are computed, we can fetch those starting points in each sub-domain. The algorithm can be modified to merge the current best tuples from each sub-domain and maintain a global neighboring block list (See Section 3.2), which determines the next candidate block for retrieving tuples.

6.2.2 Variations of Ranking Cube

We discuss some possible variations of ranking cube. We have used equi-depth partitioning to build the ranking cube. The proposed methods can also be combined with other partitioning strategies. For example, a multi-dimensional partitioning [20] recursively partitions the data domain, one dimension at a time, into bins enclosing the same number of tuples. Each multi-dimensional bin can be considered as a base block. We can still use the concept of pseudo block by merging sf_i number of consecutive base blocks in each dimension i, where sf_i is the scale factor (See Section 3) of dimension i. The query algorithm remains the same. The trade-off is that we need more space for the meta information of the base block partitioning.

To construct ranking fragments, we used a simple grouping method. There are many other criteria to group the selection dimensions. For example, if the workload (*i.e.*, query history) is available, one can compute the combination of dimensions that are frequently used in queries and materializing ranking fragments on those dimension combinations. Another criterion is to use the cardinalities of selection dimensions. If a dimension has large cardinality, further combining this dimension with other dimensions may not be useful, since the number of tuples in each cell will be too small.

6.2.3 ID List Compression

The *tid* list in each block can be compressed, such that each block contains more *tid*s. As the result, the system will retrieve less number of blocks for evaluating a ranked query. One compression method is the bitmap indexing [2, 6]. In many applications, the cardinalities of selection dimensions are small. For example, in the used car database, the majority of selection dimensions only have 2 possible values, *e.g.*, whether it has power window, sunroof, and so on. The bitmap indexing can be used to compress the *tid* lists in the ranking cube and improve the space usage. Furthermore, the merge operation in ranking fragments can be performed much faster using the bit-AND operation than the standard merge-intersect operation.

Another compression method of the *tid*-lists come from information retrieval [24]. The main observation is that the numbers in the *tid*-list are stored in ascending order. Thus, it would be possible to store a list of *tid* difference instead of the actual numbers. The insight is that the largest value in the difference list may be bounded, and it maybe possible to store them using less than the standard 32 bits of an integer.

6.2.4 High Ranking Dimensions

In this paper, we assume that the number of ranking dimensions is not large. In some applications where more ranking dimensions are involved, we can construct a variant of ranking fragments. Similar to our proposal on handling high selection dimensions, we can partition the ranking dimensions into several groups. The ranking fragments can be assembled by picking one group from selection dimensions and one group from ranking dimensions. If a query falls into two ranking fragments with different ranking groups, the query processing algorithm can be extended as follows. First, the starting base blocks are found in each ranking fragments and the *tid* lists are merged as we did in Section 4. The system then examine the neighboring blocks in each fragment and compute the best combination of the neighboring blocks as the next candidate blocks. This procedure repeats until the stop condition is satisfied.

7. CONCLUSIONS

To efficient process top-k queries with multi-dimensional selections, we proposed a novel rank-aware cube structure which is capable to simultaneously handle ranked queries and multi-dimensional selections. Based on the ranking cube, we develop a progressive query processing algorithm. We further extend the ranking cube to ranking fragments, which is especially useful for high dimensional data. Our experimental results show that the proposed methods significantly improve the query performance over the previous approaches.

Acknowledgement. We would like to thank Yijin Zhen for his valuable comments and discussions.

8. REFERENCES

[1] 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.

[2] Sihem Amer-Yahia and Theodore Johnson. Optimizing queries on compressed bitmaps. *Proceedings of 2000 International Conference on Very Large Data Bases (VLDB'00)*, pages 329–338, 2000.

[3] Stefan Berchtold, Christian Böhm, Daniel A. Keim, Hans-Peter Kriegel, and Xiaowei Xu. Optimal multidimensional query processing using tree striping. *Proceedings of 2000 International Conference on Data Warehousing and Knowledge Discovery (DaWaK'00)*, pages 244–257, 2000.

[4] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Transactions on Database Systems*, 27:153–187, 2002.

[5] Michael J. Carey and Donald Kossmann. On saying "Enough already!" in SQL. *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*, pages 219–230, 1997.

[6] Chee Yong Chan and Yannis E. Ioannidis. Bitmap index design and evaluation. *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*, pages 355–366, 1998.

[7] Kevin Chen-Chuan Chang and Seung-Won Hwang. Minimal probing: Supporting expensive predicates for top-k queries. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*, pages 346–357, 2002.

[8] 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.

[9] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR technologies: What is the sound of one hand clapping? *Proceedings of Second Biennial Conference on Innovative Data Systems Research (CIDR'05)*, pages 1–12, 2005.

[10] Bee-Chung Chen, Lei Chen, Yi Lin, and Raghu Ramakrishnan. Prediction cubes. *Proceedings of 2005 International Conference on Very Large Data Bases (VLDB'05)*, pages 982–993, 2005.

[11] Yixin Chen, Guozhu Dong, Jiawei Han, Benjamin W. Wah, and Jianyong Wang. Multi-dimensional regression analysis of time-series data streams. *Proceedings of 2002 International Conference on Very Large Data Bases (VLDB'02)*, pages 323–334, 2002.

[12] S. Chaudhuri and U. Dayal. An overview of data warehousing and data cube. *SIGMOD Record*, 26:65–74, 1997.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Ihab F. Ilyas, Rahul Shah, Walid G. Aref, Jeffrey Scott Vitter, and Ahmed K. Elmagarmid. Rank-aware query optimization. *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 203–214, 2004.

[18] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. RankSQL: Query algebra and optimization for relational top-k queries. *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD'05)*, pages 131–142, 2005.

[19] X. Li, J. Han, and H. Gonzalez. High-dimensional OLAP: A minimal cubing approach. *Proceedings of 2004 International Conference on Very Large Data Bases (VLDB'04)*, pages 528–539, 2004.

[20] M. Muralikrishna and David J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD'88)*, pages 28–36, 1988.

[21] Patrick E. O'Neil and Dallan Quass. Improved query performance with variant indexes. *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*, pages 38–49, 1997.

[22] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD'84)*, pages 256–276, 1984.

[23] W. Rudin. Principles of mathematical analysis, 3rd ed. *New York: McGraw-Hill*, 1976.

[24] A. Singhal. Modern information retrieval: A brief overview. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 24(4):35–43, 2001.