Inverse Ranking Queries

Chengkai Li
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Avenue
Urbana, IL 61801
cli@uiuc.edu

Kevin Chen-Chuan Chang
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Avenue
Urbana, IL 61801
kcchang@cs.uiuc.edu

Ihab F. Ilyas
School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada N2L 3G1
ilyas@uwaterloo.ca

ABSTRACT

In this paper, we identify a novel and interesting type of queries, *inverse ranking queries*, which return the ranks of certain tuples among the context given in the queries. Inverse ranking queries are very useful in non-traditional data retrieval and exploration. They provide a mechanism to quickly identify where some tuples stand within the context in question. While important, such queries are so far unknown to the community, therefore there does not exist efficient evaluation methods for them. In this paper, we first extend the SQL language to express inverse ranking queries. We then propose a general partition-based framework for processing such queries. The framework embraces implementation methods that exploit common data structures in databases, as well as a novel method that utilizes a new data structure based on bitmap index. We analytically investigate the pros and cons of these methods, according to a preliminary cost model. We further conduct experiments to empirically compare them. The results show that our new algorithms can be significantly more efficient than a straightforward materialize-then-sort approach.

To the best of our knowledge, ours is the first work that proposes and thoroughly studies the inverse ranking queries. Moreover, interestingly the dual form of such queries, quantile queries, is also very useful. Quantile queries obtain the tuples ranked at any given positions. They provide a fast sketch of the data and also give users a quick feeling about the ranking criteria, thus form the basis for further analytical queries. Furthermore, certain quantile points have significant statistical meaning. Although this paper focuses on inverse ranking queries, the proposed framework and algorithms can be easily adapted to process quantile queries as well, due to the duality between these two kinds of queries.

1. INTRODUCTION

The ubiquitous usage of databases for managing structured data, compounded with the expanded reach of the Internet to end users, has brought forward new scenarios of data *retrieval* and *exploration*. Users often want to express non-traditional fuzzy retrieval with soft criteria (e.g., similarity, relevance, preference, *etc.*), in contrast to Boolean queries, and to explore what choices are available and how they match the query criteria. Such retrieval and exploration are critical in many emerging applications, including E-commerce, multimedia retrieval, DB/IR integration, decision support, *etc.* While successful in business settings, conventional database management systems (DBMSs) have become increasingly inadequate for such new scenarios. As an important first step, many recent works have focused on ranking (top-k) queries [15, 7, 14, 9, 2, 21, 10, 23, 20, 33, 13], for finding the best-matching k answers in databases according to user-specified ranking functions.

In this paper, we identify a novel and interesting type of *inverse ranking queries*, symmetrical to ranking. Such a query obtains the rank of a data record within a certain context given in the query. Inverse ranking queries are useful in many places. For instance, a credit card company may be interested in the standing of a new customer among her peers, in order to determine her credit line. As another example, we may want to compare a newborn baby to others with respect to their heights, weights, *etc.* While ranking has recently gained significant attention from the research community, inverse ranking query has not been studied so far, in contrast to its usefulness. This paper thus proposes and defines the query model and SQL language extension to enable expressing such queries. To the best of our knowledge, this is the first work that studies inverse ranking queries.

With the current query processing techniques, a straightforward *exhaustive* approach of processing inverse ranking queries is to fully materialize the results of a Boolean query, *i.e.*, the context of the ranking, and then count the number of tuples whose ranking scores are higher than the score of the object in question. The rank of the given object is thus obtained as a post-processing step. Such an exhaustive approach can be inefficient, as the query only asks for the rank of a certain tuple, while the full Boolean results are indeed made. Observing the symmetricity between inverse ranking queries and top-k queries, it may appear that the recently developed top-k query algorithms can be adopted. However, top-k algorithms are explicitly optimized for retrieving very small number (k) of top answers. When k is relatively large, the performance of these algorithms become even worse than the straightforward approach [21, 23]. Unfortunately, the tuple in question in an inverse ranking query can be ranked anywhere such as in the middle or at the bottom, corresponding to fairly large k. Therefore we must look for novel and efficient processing techniques.

This paper designs a *partition-and-prune* framework for processing inverse ranking queries. The framework starts by partitioning the space of tuples into buckets. The upper and lower bounds of ranking scores for tuples within each bucket are derived. These bounds determine which are the candidate buckets whose tuples rank near the given tuple in the query. After computing their cardinalities (number of tuples), the non-candidate buckets can be safely pruned because, for any such bucket, its tuples are all ranked higher (lower) than the query tuple. Therefore we only need to retrieve the tuples in the candidate buckets in order to obtain the ranking position of the query tuple. To realize this general framework, we introduce several partition schemes and implementation methods. Some of these methods exploit common data structures in database systems, while some others utilize bitmap index built over ranking functions. Our experimental study shows that our algorithms can be significantly more efficient than the exhaustive method.

While inverse ranking query is compelling by itself, we find that its dual form, *quantile query*, is also important. A quantile query returns the results at certain ranking positions according to a ranking function. More specifically, the q quantile of a given set of data is the value v_q such that the fraction q of the data are higher than v_q. Quantile queries can be useful in non-traditional data retrieval and exploration since they provide a mechanism of fast-forwarding over the data, as an analogy to such functionalities in video and audio equipments. They locate the quantiles as a sketch of the query results, which give users a quick feeling about both the data and the ranking function, and thus help in continuously exploring the data. Moreover, quantile points have significant statistical meanings. For example, the 50% quantile is the mean of a dataset; the *quantile-quantile* (*q-q*) plot is a graphical method used in statistics to compare the distributions of two sets of data and determine if they come from populations with a common distribution.

This paper is also the first that studies such quantile queries in the general context of querying databases. Previous works on computing quantiles [1, 3, 17, 25, 4, 12, 11, 36] focus on the quantiles for a set of data values, whereas we study the quantiles among database records that are ranked by functions combining multiple criteria (attributes). More importantly, the context of ranking in this paper is general database queries with Boolean conditions (selections and joins), which are not considered by previous works.

The duality between inverse ranking queries and quantile queries allows us to apply the same framework and similar algorithms in processing them. However, we focus on inverse ranking queries in the paper, and do not discuss such adaptation.

In summary, this paper makes the following contributions:

- **Concept: inverse ranking queries and quantile queries.** We identify the query models and propose the SQL extensions for defining these queries. To the best of our knowledge, ours is the first in the literature to study inverse ranking queries, and the first to investigate quantile queries in general database query context.
- **Framework: partition-and-prune approach.** We design this general framework to avoid costly exhaustive approach. We also develop a preliminary cost model to assist the analytical comparisons of various implementation methods for the framework.
- **Implementations.** We develop several methods to instantiate the framework, utilizing existing and new data structures. We analyze the pros and cons of these methods, based on the cost model. Our empirically study verify that some of our methods can be significantly more efficient than the exhaustive approach.

The rest of the paper is organized as follows. In Section 2, we introduce the SQL extension to define inverse ranking and quantile queries and illustrate the example queries. Section 3 presents the partition-and-prune framework, focusing on inverse ranking queries. Several implementation methods for the framework are introduced in Section 4. We experimentally evaluate the proposed framework and algorithms in Section 5. Section 6 reviews related work. Finally, we conclude the paper in Section 7.

2. DEFINING INVERSE RANKING QUERIES AND QUANTILE QUERIES

In this section, we propose extensions to SQL language for expressing inverse ranking and quantile queries, and use examples to motivate their applications.

To specify an inverse ranking query, we overload the OLAP function *rank*() in SQL[1], as shown below:

SELECT ..., *rank*() **IN** (**SELECT** ...
  FROM $R_1, ..., R_n$
  WHERE $\mathcal{B}(c_1, \ldots, c_j)$)
FROM $R'_1, ..., R'_h$
WHERE $\mathcal{B}'(c'_1, \ldots, c'_l)$
ORDER BY $\mathcal{F}(p_1, \ldots, p_m)$

In an inverse ranking query Q, the product of the base relations $R'_1 \times \ldots \times R'_h$, filtered by a *Boolean function* $\mathcal{B}'(c'_1, \ldots, c'_l)$ (*e.g.*, $\mathcal{B}'{=}c'_1 \wedge c'_2 \wedge c'_3$), constitutes the *query tuples*. Following *rank*() **IN**, another Boolean function $\mathcal{B}(c_1, \ldots, c_j)$, applied over $R_1 \times \ldots \times R_n$, supplies the *context tuples*. A *ranking function* $\mathcal{F}$ over the *ranking attributes* $p_1, \ldots, p_m$ (*e.g.*, $\mathcal{F}{=}p_1{+}p_2{+}p_3$) computes the *ranking scores* of context and query tuples. The query returns the ranks of the query tuples among the context tuples, by the descending order of their scores[2]. Note that in order to make the ranking function applicable, the schema of both context and query tuples should contain the ranking attributes.

Formally, Q returns query tuples $R_{\mathcal{B}'}{=}\sigma_{\mathcal{B}'(c'_1,\ldots,c'_l)}$ $(R'_1 \times \cdots \times R'_h)$ together with their ranks, determined as follows. Each tuple t has a *ranking score* $\mathcal{F}[t]$. For a query tuple $t_q{\in}R_{\mathcal{B}'}$, its rank is the number of context tuples t_c that have higher scores than t_q (plus 1), *i.e.*, $rank(t_q){=}1{+}|\{t_c|\mathcal{F}[t_c]{>}\mathcal{F}[t_q], t_c{\in}R_{\mathcal{B}}\}|$, where $R_{\mathcal{B}}{=}\sigma_{\mathcal{B}(c_1,\ldots,c_j)}(R_1 \times \cdots \times R_n)$ is the *context relation*. When there are ties in scores, an arbitrary *deterministic* "tie-breaker" function can be used to determine an order, *e.g.*, by tuple IDs.

The aforementioned query obtains query tuples by Boolean selection and join conditions, thus it requires the existence of query tuples in the Boolean results. However, we may be interested in the ranks of virtual tuples that do not necessarily exist. Therefore we propose the following alternative syntax that requests the rank of a virtual tuple $p_1 = v_1, \ldots, p_m = v_m$.

SELECT ..., *rank*() **IN** (**SELECT** ...
  FROM $R_1, ..., R_n$
  WHERE $\mathcal{B}(c_1, \ldots, c_j)$)
VALUES $(p_1 = v_1, \ldots, p_m = v_m)$
ORDER BY $\mathcal{F}(p_1, \ldots, p_m)$

Example 1: Consider a credit card company. To decide whether to approve the increase of credit line upon the request from a customer (with customer id 1001), the following query gives the rank of the customer among the customers in the same area. Various ways can be explored in computing the ranking scores. For example, a weighted average of the customer's income, age, and credit history is used in the query.

select *cid*, *rank*() **in** (**select** *
  from $Customer$
  where *zipcode*=12345)
from $Customer$
where *cid*=1001
order by $w_1{\times}income{+}w_2{\times}age{+}w_3{\times}credit$

[1]OLAP functions, introduced in SQL99 and supported by major DBMSs, are for different purpose than inverse ranking.

[2]We assume **Order By Asc|Desc** uses descending order (**Desc**) as default, although **Asc** is the default in some systems.

Alternatively, we might want to use the following query to determine the rank of a user who is not an existing customer.

select *cid*, *rank*() **in** (**select** *
from $Customer$
where $zipcode$=12345)
values ($income$=50000, age=30, $credit$=600)
order by $w_1 \times income + w_2 \times age + w_3 \times credit$ ■

To specify a quantile query, we extend the syntax of SQL language by adding a **QUANTILES AT** clause, as shown below:

SELECT ...
FROM R_1, ..., R_n
WHERE $\mathcal{B}(c_1, \ldots, c_j)$
ORDER BY $\mathcal{F}(p_1, \ldots, p_m)$
QUANTILES AT q_1, ..., q_k

The semantics of such a query Q is that, among the filtered Boolean results $R_\mathcal{B}$, ranked by $\mathcal{F}(p_1, \ldots, p_m)$, those tuples ranked at the given quantile positions q_1, ..., q_k are returned. Formally, Q returns k tuples[3] $\{t_1, \ldots, t_k\}$ from $R_\mathcal{B}$ at the quantiles q_1, ..., q_k, such that $rank(t_i)=\lceil q_i \times |R_\mathcal{B}| \rceil$ (for $0 \leq q_i \leq 1$, representing percentile) or $rank(t_i)=\lceil q_i \rceil$ (for $q_i > 1$, representing absolute ranking position).

Example 2: Consider a real estate company that is interested in identifying their representative houses at several ranking positions, according to the perspective of a certain type of customers. The houses are ranked by size, price, and so on. The quantile query is shown below.

select $*$, *size*/*price* **as** *score*
from $Houses$
where $zipcode$=12345
order by $score$
quantiles at 0.1, 0.5, 0.7, 0.95 ■

3. PARTITION-AND-PRUNE FRAMEWORK

In this section, we present a general framework for processing inverse ranking queries. To answer such queries, with the ranking scores of the given query tuples, we must locate where the query tuples stand among the context tuples. As discussed in Section 1, the problems with the straightforward exhaustive approach is that it fully materializes the context tuples. Therefore the key of a more efficient solution lies in avoiding such full materialization. To be more specific, we want to prune irrelevant context tuples and quickly zoom into the regions containing tuples with scores close to the query tuples.

The framework is simple and intuitive. We partition the space of tuples into buckets and compute the upper-bound and lower-bound of ranking scores of the tuples within each bucket. These bounds classify the buckets into three categories, with respect a given query tuple. The buckets with lower-bounds higher than the score of the query tuple contain context tuples ranked higher than the query tuple; the buckets with upper-bounds lower than the query tuple score contain lower-ranked context tuples; and the context tuples in the rest of the buckets, the *candidate tuples*, may be ranked higher or lower than the query tuple. Therefore by counting the cardinalities (number of tuples) of the buckets, we know how many context tuples are guaranteed to be ranked higher (lower) than the query tuple, and we only need to look up the scores of the candidate tuples to obtain the rank of the query tuple. We illustrate the idea in Example 3, as our running example.

[3]More rigorously, it returns $min(k, |R_\mathcal{B}|)$ tuples.

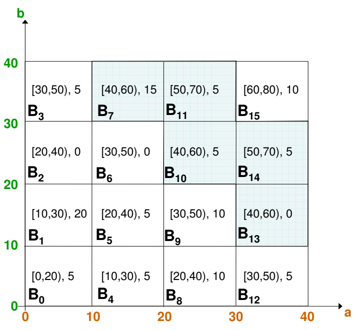


Figure 1: Buckets in a 2-dimensional tuple space.

Example 3: Consider the following inverse ranking query, which asks for the ranks of those tuples in R with a=35 and b=20, ranked by a+b.

select *, rank() **in** (**select** * **from** R)
from R
where a=35 **and** b=20
order by a+b

Figure 1 shows the two-dimensional tuple space (on attributes a and b) partitioned into 16 buckets. The ranges of a and b for each bucket are shown. For instance, the bucket in the left lowermost corner (B_0) has ranges $0 \leq a < 10$ and $0 \leq b < 10$. The ranges on a and b determine the upper-bound and lower-bound of tuple scores for buckets. We show the bounds and the cardinality inside each bucket. For instance, B_0 contains 5 tuples, which have the lower-bound and upper-bound score 0 and 20, respectively.

Among these buckets, the shaded ones are candidate buckets and others are pruned buckets. Bucket B_{15} is pruned because the scores of tuples inside it are at least 60, thus higher than the score of the query tuples, 35+20=55. The 10 left lower buckets are also pruned because their tuple scores are lower than 55. The tuples in the candidate buckets may score higher or lower than 55. There are 10 tuples in B_{15} and totally 30 tuples in the candidate buckets. Therefore the ranks of the query tuples are between 11 and 40. To get their ranks, we must obtain the tuples in the candidate buckets and resolve their orders to the query tuples. ■

Based on the intuition from the above example, below we formally present the framework. Section 3.1 defines the partitioning and pruning of tuple space, which are the basis of the procedural general algorithm in Section 3.2. Section 3.3 discusses the cost model for the algorithm. The cost model helps us in designing and analyzing various schemes of partitioning in Section 3.4, and is the guideline in realizing the partitioning schemes and the algorithm, in Section 4.

Throughout the discussion, we assume there is only one single table, without Boolean conditions over context tuples. In other words, we assume the context relation $R_\mathcal{B}$ is simply one base table. In Section 4, we discuss how to extend the techniques to handle join and selection conditions. Moreover, we will not discuss how to obtain the query tuples $R_{\mathcal{B}'}$. Note that such tuples are given when the query is about virtual tuples, using **VALUES**. In other cases, we need to obtain the query tuples and cannot avoid the overhead. Certainly there are situations when the cost of obtaining the query tuples themselves dominates that of getting their ranks.

3.1 Tuple Space Partitioning and Pruning

Definition 1 (Partition, Bucket, Constraint): A relation, $R(a_1, a_2, \ldots)=\{t_1, t_2, \ldots\}$, is a set of tuples $\{t_1, t_2, \ldots\}$ with the schema

A=$\{a_1, a_2, \ldots\}$. A *partition* $\mathcal{P}_\mathcal{R}$=$\{b_1, b_2, \ldots\}$ is a set of subsets of R such that $\cup b_i$=R, $b_i \neq \phi$, and $b_i \cap b_j$=ϕ, $\forall b_i, b_j \in P_R$. Each subset b_i is a *bucket*. Given a bucket b_i, its cardinality $|b_i|$ is the number of tuples belonging to b_i. That is, b_i=$\{t_{i_1}, \ldots, t_{i_{|b_i|}}\}$, where $t_{i_j} \in R, \forall 1 \leq j \leq |b_i|$. Each bucket b_i is associated with a set of constraints C_i=$\{c_{i_1}, c_{i_2}, \ldots\}$. Each constraint is of the form $l \leq g(A) < u$, where $g(A)$ is a function over A. Given any tuple $t_{i_j} \in b_i$, all the constraints associated with b_i are satisfied. ■

Example 4: Continue Example 3. Figure 1 is a partition of R, containing 16 buckets. The cardinalities of the buckets are shown in Figure 1. Every bucket is associated with two constraints. For instance, bucket B_0 has constraints $\{0 \leq a < 10,\ 0 \leq b < 10\}$. All the tuples in B_0 thus have both attribute a and b in the range $[0, 10)$. Note that the constraints associated with these buckets are in the form of very simple function– a single attribute. ■

Definition 2 (Upper-Bound, Lower-Bound): Given a bucket b, an upper-bound score $\lceil b \rceil$ is a value that is larger than the highest score among tuples in b. That is, $\lceil b \rceil > \mathcal{F}[t], \forall t \in b$. Similarly, a lower-bound score $\lfloor b \rfloor$ is a value that is smaller than or equal to the lowest score among tuples in b. That is, $\lfloor b \rfloor \leq \mathcal{F}[t], \forall t \in b$. [4] [5] ■

Definition 3 (Pruned and Candidate buckets): With respect to a query tuple t_q, a bucket b is a *pruned bucket* if $\mathcal{F}[t_q]$, the score of t_q, is an upper-bound of b, or if $\mathcal{F}[t_q]$ is a lower-bound of b and there is no tuple in b with score equal to $\mathcal{F}[t_q]$. Formally, given a partition $\mathcal{P}_\mathcal{R}$, the set of pruned buckets with respect to t_q is $pruned(\mathcal{P}_\mathcal{R}, t_q)$ = $pruned^+(\mathcal{P}_\mathcal{R}, t_q) \cup pruned^-(\mathcal{P}_\mathcal{R}, t_q)$, where $pruned^+(\mathcal{P}_\mathcal{R}, t_q)$=$\{b | b \in \mathcal{P}_\mathcal{R}$ and $\mathcal{F}[t] > \mathcal{F}[t_q], \forall t \in b\}$ are the *dominating buckets* of t_q and $pruned^-(\mathcal{P}_\mathcal{R}, t_q)$=$\{b | b \in \mathcal{P}_\mathcal{R}$ and $\mathcal{F}[t] < \mathcal{F}[t_q], \forall t \in b\}$ are the *dominated buckets* of t_q. The set of candidate buckets is $candidate(\mathcal{P}_\mathcal{R}, t_q)$=$\mathcal{P}_\mathcal{R} - pruned(\mathcal{P}_\mathcal{R}, t_q)$. ■

Example 5: Continue Example 4. The bucket B_0 has constraints $\{0 \leq a < 10,\ 0 \leq b < 10\}$. Thus the tuples in B_0 can score at most 10+10=20 (without equality), and as low as 0, *i.e.*, $\lceil B_0 \rceil$=20 and $\lfloor B_0 \rfloor$=0.[6] Similarly, we can obtain the bounds of other buckets. The query tuple has score 55, therefore the white buckets in Figure 1 are pruned buckets and the shaded buckets are candidate buckets, based on Definition 3. ■

In determining the rank of the query tuple, we can safely prune the pruned buckets and we only need to look up the tuples in the candidate buckets. The idea is already illustrated in Example 3. More formally, we have the following Property 1. The straightforward proof is omitted.

Property 1 (Bucket Pruning Property): Given a relation R and its partition $\mathcal{P}_\mathcal{R}$, the rank of a query tuple t_q is
rank(t_q)=1 + $\sum_{b \in pruned^+(\mathcal{P}_\mathcal{R}, t_q)} |b|$ + $|\{t_c | \mathcal{F}[t_c] > \mathcal{F}[t_q] \wedge t_c \in b\ s.t.\ b \in candidate\ (\mathcal{P}_\mathcal{R}, t_q)\}|$. ■

3.2 General Algorithm

Based on Property 1, we design the general algorithm for answering inverse ranking queries, as outlined in Figure 2. The algorithm takes the following steps in sequence:

[4]Without loss of generality, we require $\lceil b \rceil$ to be open-ended while $\lfloor b \rfloor$ to be close-ended. Correspondingly the constraints in Definition 1 are left-end closed and right-end open.

[5]Note that there is an infinite number of upper-bounds and lower-bounds for any bucket.

[6]More strictly, $\lceil B_0 \rceil$=20 means 20 is one known upper-bound for b. Similar statement applies for $\lfloor B_0 \rfloor$.

```
Procedure Partition-and-Prune Inverse Ranking
    /* relation: $R$, with schem $A$ */
    /* partition: $\mathcal{P}_\mathcal{R}$ */
    /* ranking function: $\mathcal{F}(A)$ */
    /* query tuple: $t_q$ */
begin
 1: /* 1. Partitioning Space */
 2: determine the number of buckets, $n$
 3: for each bucket $b_i$ do
 4:     determine constraints $C_i$=$\{c_{i_1}, c_{i_2}, \ldots\}$, where $c_{i_j}$ is:
 5:         $l_{i_j} \leq g_i(A) \leq u_{i_j}$
 6:
 7: /* 2. Deriving Bounds */
 8: for each bucket $b_i$ do
 9:     /* solve the following optimization problem */
10:     $\lceil b_i \rceil \leftarrow$ the value maximizes $\mathcal{F}(A)$ in $b_i$
11:     $\lfloor b_i \rfloor \leftarrow$ the value minimizes $\mathcal{F}(A)$ in $b_i$
12:
13: /* 3. Computing Cardinalities */
14: for each bucket $b_i$ do
15:     $|b_i| \leftarrow$ compute the number of tuples in $b_i$
16:
17: /* 4. Classifying Buckets */
18: $pruned^-(\mathcal{P}_\mathcal{R}, t_q) \leftarrow \phi$ /* dominated buckets */
19: $pruned^+(\mathcal{P}_\mathcal{R}, t_q) \leftarrow \phi$ /* dominating buckets */
20: $candidate(\mathcal{P}_\mathcal{R}, t_q) \leftarrow \phi$ /* candidate buckets */
21: for each bucket $b_i$ do
22:     if $\lceil b_i \rceil \leq \mathcal{F}[t_q]$ then
23:         $pruned^-(\mathcal{P}_\mathcal{R}, t_q) \leftarrow pruned^-(\mathcal{P}_\mathcal{R}, t_q) \cup b_i$
24:     else if $\lfloor b_i \rfloor > \mathcal{F}[t_q]$ then
25:         $pruned^+(\mathcal{P}_\mathcal{R}, t_q) \leftarrow pruned^+(\mathcal{P}_\mathcal{R}, t_q) \cup b_i$
26:     else
27:         $candidate(\mathcal{P}_\mathcal{R}, t_q) \leftarrow candidate(\mathcal{P}_\mathcal{R}, t_q) \cup b_i$
28:
29: /* 5. Retrieving Candidates */
30: $R' \leftarrow \cup_{b_i \in candidate(\mathcal{P}_\mathcal{R}, t_q)} b_i$ /* candidate tuples */
31: retrieve tuples in $R'$
32: $rank_{R'}(t_q) \leftarrow$ the rank of $t_q$ in $R'$
33: $rank(t_q) \leftarrow rank_{R'}(t_q)$ + $\sum_{b \in pruned^+(\mathcal{P}_\mathcal{R}, t_q)} |b|$
34: return $t$
end
```

Figure 2: The outline of the algorithm.

1. Partitioning Space: The algorithm partitions the tuple space into buckets. It needs to decide the number of buckets and the constraints associated with each bucket. The constraints directly determine the geometrical shape and area of a bucket.

2. Deriving Bounds: The algorithm derives the upper-bound and lower-bound of every bucket, based on the associated constraints. For a set of general constraints, deriving the bounds of a ranking function is a *nonlinear programming* (NLP) problem. While general NLP problem is very hard, there are methods for special cases [5].

In this paper, we concentrate on *monotonic* ranking functions, as commonly studied in top-k queries (*e.g.*, [15]). Examples of such functions include *sum*, *weighted average*, *Lp-norm distance* such as *Manhattan* and *Euclidean distance*, *etc.* With single-attribute constraints in the form of $l \leq a < u$, the bounds of such monotonic ranking function can be straightforwardly determined by the ranges on the attributes in the bucket. Therefore given a partitioning scheme using only single-attribute constraints, the algorithm can handle any monotonic ranking functions.

More specifically, deriving the bounds becomes a *linearly constrained optimization* problem when all the constraints are linear functions over the attributes, and further a *linear programming* (LP) problem [6] when the ranking function is a linear function as well. There are well-studied algorithms for solving LP problems, such as the Simplex method [6]. Therefore given a partitioning scheme using linear constraints, the algorithm can process linear ranking functions, a subset of monotonic functions. [7] [8]

3. Computing Cardinalities: The algorithm computes the cardinality of every bucket according to the associated constraints.

4. Classifying Buckets: The bounds from step 2 and cardinalities from step 3 are used to identify the pruned and candidate buckets, following Definition 3.

5. Retrieving Candidates: The algorithm retrieves the tuples in the candidate buckets, evaluates their scores, and obtains the ranks of the query tuples.

Among the five steps, step 2 and 4 are described above and shown in Figure 2. They are not further discussed. Step 1 is the basis of the algorithm, since the partitioning scheme determines what type of ranking functions can be handled and which implementation methods for other steps are applicable. An appropriate partitioning scheme is thus key to the efficiency of the algorithm. In Section 3.3 we describe an analytical cost model of the algorithm. Guided by the analysis, we discuss several partitioning schemes in Section 3.4. There are different methods in implementing the partitioning scheme and thus the step 3 and 5. Such choices directly affect the efficiency of the algorithm. We present implementation details in Section 4.

3.3 Cost Model

Given the variety of implementations for the above algorithm steps, there exists an optimization space in choosing an efficient evaluation method for inverse ranking queries. The primitive cost model in this section is for the purpose of analyzing and comparing the choices in realizing our framework. In query optimization, a cost model is critical for estimating the costs of query plans. However, a complete cost model for inverse ranking queries, incorporated into query optimizer, is out of our focus in this paper.

The cost model has the following components:

Cost factors: The cost of our algorithm is determined by several factors, which are the partition, the data distribution, the data size, and the query.

Cost parameters: The cost is a function of several important parameters, including the number of buckets ($|\mathcal{P}_\mathcal{R}|$), the score bounds of the buckets ($\lfloor b \rfloor$ and $\lceil b \rceil$), the cardinalities of the buckets ($|b|$), and the candidate buckets ($candidate(\mathcal{P}_\mathcal{R}, t_q)$). These cost parameters are determined by the cost factors above. Specifically, the partition determines the number of buckets and the bounds (constraints) of the buckets, the data distribution and data size determine the cardinalities, and the query determines the candidate buckets, together with the partition, data distribution and size.

Cost formula: The cost formula in terms of time, as shown below, is the sum of the CPU cost, the I/O cost for obtaining cardinalities (step 3 in Section 3.2) and the I/O cost for retrieving candidate tuples (step 5).

$$\mathcal{C} = \mathcal{C}_{CPU} + \mathcal{C}3_{I/O} + \mathcal{C}5_{I/O} \tag{1}$$

Conventional query algorithms are I/O-bound rather than CPU-bound, therefore the common practice in investigating the costs of query plans is to focus on disk I/O cost. However, we shall see that some of the methods in Section 4 involve CPU costs that cannot be ignored. Among the five steps in Figure 2, step 1, 2, and 4 do not involve disk I/O. To obtain the cardinalities of buckets in step 3, some of our methods require I/O access to auxiliary data structures. For step 5, the retrieving of candidate tuples involves disk access.

Nevertheless, in most cases, the most significant component in the above formula is the disk I/O cost of step 5, for which a good metric is the number of candidate tuples. That is,

$$\mathcal{C}5_{I/O} = f_{retrieve} \times \sum_{b \in candidate(\mathcal{P}_\mathcal{R}, t_q)} |b| \tag{2}$$

where $f_{retrieve}$ is a factor. In principle the more candidate tuples, the higher cost, although the exact $f_{retrieve}$ depends on the specific method of retrieving tuples.

Figure 3 summarizes the relationships among the cost parameters, in determining the number of candidate tuples. We use the up-arrow and down-arrow to represent "increase in amount" and "decrease in amount", respectively. *First*, the number of candidate tuples is directly determined by two cost parameters, the bounds and the cardinalities. To be more specific, the more tuples in each candidate bucket, the more candidate tuples; and the bigger interval between the upper-bound and the lower-bound ($\lceil b \rceil - \lfloor b \rfloor$) of each bucket, the more candidate buckets (since the chance of subsuming the query tuple score is bigger), thus the more candidate tuples.

Second, the cardinalities and bounds are determined by the partition, *i.e.*, the number of buckets and the constraints of each bucket. The relationship between the number of buckets and the cardinalities/bounds is clear. The more buckets, the less tuples in each bucket, and the smaller intervals between the upper and lower bounds. This seems to indicate that we should have as many buckets as possible, since that can result in less candidate tuples. However, with more buckets, the cost of constructing the buckets and computing their cardinalities ($\mathcal{C}3_{I/O}$ and part of $\mathcal{C}_{CPU}$) can be significant.

In addition to the number of buckets, the constraints also determine the bounds. This relationship is illustrated by the following example.

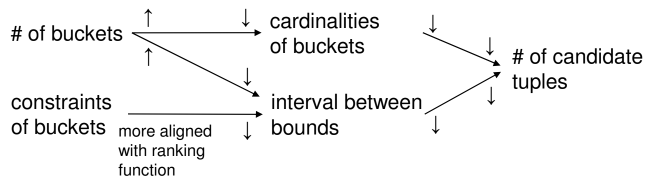
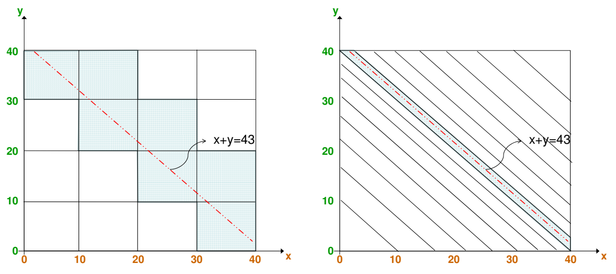


Figure 3: The relationship among cost parameters.



(a) (b)

Figure 4: The relationship between constraints and bounds.

Example 6: Figure 4 shows a two-dimensional space over attributes x and y of relation R. Consider ranking function x+y. We are looking for the rank of a tuple with score 43. The tuples with the same ranking score locate on a unique contour line, one for each score value. For instance, the contour line for x+y=43 is shown as a

[7]Note that the above single-attribute constraint is an extreme case of linear constraint.

[8]Linear function such as $2p_1 - 3p_2$ is monotonic on p_1 and $-p_2$.

dashed line in Figure 4(a) and (b).

Different constraints can result in very different bounds. Figure 4(a) and (b) show two partitions of the same space, where the solid lines are the boundaries of the buckets. The partition in (a) uses constraints of the form $\{x_1 \leq x < x_2,\ y_1 \leq y < y_2\}$, while the partition in (b) has constraints of the form $\{l \leq x+y < u\}$, *i.e.*, the constraints are parallel to the contour lines of the ranking function. Both (a) and (b) partition the space into 16 buckets with the same area size, thus roughly the same cardinality under the assumption of uniform data distribution. The figures show that, although containing about the same number of tuples, the buckets in (b) have much smaller intervals between bounds than the buckets in (a) have. Therefore there are 7 candidate buckets (in shade) in (a), while (b) only has 1 candidate bucket. ■

The above example illustrates that which constraint results in the smallest intervals between bounds depends on the ranking function itself. For instance, the contour lines for $2x+y$ have different slope than that in Figure 4, thus the constraints parallel to the contour lines are also different.

3.4 Partitioning Schemes

The analysis in Section 3.3 indicates that the most significant cost component, the number of candidate tuples, is determined by the partitioning scheme, which consists of the number of buckets and the constraints, by definition. The constraints of the buckets specify the way to partition, while the number of buckets indicates the granularity of the partition in that way. Below we present several partitioning schemes, *i.e.*, various types of constraints. These schemes are implemented by the methods in Section 4.

Single-Attribute Constraints:
A straightforward approach of partitioning is to associate with the buckets the simplest constraints, which are intervals (ranges) over individual attributes. That is, a constraint has the form $l \leq a < u$, where a is one attribute, and l and u are some constant values. In this partition, the boundaries between buckets are parallel to the dimensions, *i.e.*, attributes. One advantage of using single-attribute constraints is that they can support any monotonic functions on these attributes. Moreover, it is easy to conduct satisfaction test for these constraints. For instance, a B-tree on a may be used in obtaining those tuples satisfying constraint $l \leq a < u$.

Function Constraints:
Partitioning based on single-attribute constraints can be sub-optimal, depending on the ranking functions. As discussed in Section 3.3, the constraints should be aligned with the contour lines of the ranking function, in order to achieve small number of candidate tuples. Following this intuition, we propose a partitioning scheme based on constraints as functions.

In this scheme, each constraint is of the form $l \leq g < u$, where g is a linear function. Given a linear ranking function f, if g is "close" to f (in other words, g is aligned with f), the number of candidate buckets and tuples can be small. Such closeness can be measured by the angle between f and g, *i.e.*, the cosine similarity of their coefficients. Note that this scheme is applicable when we consider only linear ranking functions and linear constraints. The problem of computing the bounds of such ranking functions, given the constraints, is a linear programming problem, as we discussed in Section 3.2.

Satisfaction test for such function constraints requires us to build auxiliary data structures. That is, we need to build an index over g (instead of single attribute a).

Workload-Based Function Constraints:
For the above scheme using function constraints, in order to achieve small number of candidate tuples, the function g used in the constraint should be close to f. In other words, we must have indices built for various different g, so that among them we can find one that is close to the dynamic ranking function f in the query. However, as the number of attributes involved in ranking functions increases, the necessary number of indices for the g may become prohibitively large.

Our idea in tackling this challenge is use query workload to guide the selection of functions g to build index for. The indexed functions are chosen such that they are "close" to many previous queries according to the query workload. With the reasonable expectation that future queries share the same characteristics with the workload, the indexed functions can capture many queries in the future. [9]

4. IMPLEMENTATION METHODS

In this section, we present several implementation methods of the partition-and-prune framework in Section 3. These methods utilize a variety of data structures in DBMS including histogram, B-tree, multi-dimensional index, and bitmap index. When introducing each method, we first describe the details of partitioning space (step 1), computing cardinalities (step 3), and retrieving candidates (step 5), respectively. Then we analyze the pros and cons of the method. We first introduce those methods realizing partitioning schemes based on single-attribute constraints (Section 4.1), then propose methods for schemes using function constraints and workload-based function constraints (Section 4.2). Finally, we discuss how to deal with Boolean selection and join conditions in the queries (Section 4.3).

4.1 Implementations of Partitioning by Single-Attribute Constraints

4.1.1 *Using Multi-Dimensional Histogram*

Histogram is commonly used in query optimization (for selectivity estimation) and approximate query answering in database systems [22]. A multi-dimensional histogram naturally partitions the tuple space into buckets, with the cardinality of every bucket stored.

Partitioning Space: In a histogram, each bucket is defined by intervals (ranges) over individual attributes. For instance, the partition in Figure 1 can indeed be a two-dimensional histogram.

Computing Cardinalities: The advantage of using multi-dimensional histogram is that the cardinalities of buckets are pre-computed.

Retrieving Candidates: The histogram maintains cardinality information, but cannot provide access to individual tuples. Therefore, we must use SQL range queries, one for each bucket, to retrieve the tuples in the candidate buckets.

Example 7: Suppose Figure 1 is a histogram. The range queries corresponding to the candidate buckets are shown below. Note that the multiple range queries are concatenated by **UNION** in a single SQL query. An alternative is to use disjunctive conditions in the **WHERE** clause, *i.e.*, ($10 \leq a$ and $a < 20$ and $30 \leq b$ and $b < 40$) or $\cdots$ or ($30 \leq a$ and $a < 40$ and $20 \leq b$ and $b < 30$).

(**select** $*$ **from** R
where $10 \leq a$ and $a < 20$ and $30 \leq b$ and $b < 40$)
union
...

[9]By the same intuition, in other works, query workload is also used in choosing conventional indices to construct and views to materialize.

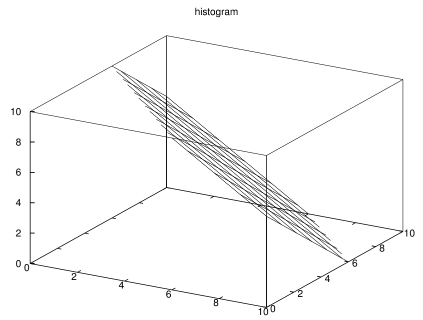


Figure 5: Inverse ranking query for a tuple with score $x+y+z=16$, using histogram.

union
(**select** $*$ **from** R
where $30 \le a$ and $a < 40$ and $20 \le b$ and $b < 30$) ■

Although using multi-dimensional histogram provides cardinality automatically, this method has serious disadvantages. *First*, the transformed SQL query is inefficient to evaluate, as the multiple range queries may require the access to the full domain of every attribute. To illustrate, consider Figure 5. Suppose the ranking function is $x+y+z$, and the query asks for the rank of a tuple with score 16. The candidate buckets must at least contain the gray plane $x+y+z=16$, which spans through the whole domain of x, y, and z, respectively. Note that multi-dimensional histogram was also used in answering *top-k* queries [9]. However, only one range query is needed for one *top-k* query, because the candidates of top k answers are located in a small area around the query point.

Second, the dimensions in the histogram may not match the attributes in the ranking function, resulting in loose upper-bound and lower-bound. The loose bounds further produce significant overlapping among the buckets' bounds, thus large number of candidate buckets. For instance, suppose the histogram in Figure 1 is used to answer another inverse ranking query with ranking function $a+b+c$, instead of $a+b$. The attribute c has domain $[0, 40)$. Since the histogram only uses a and b to partition the space, a constraint, $0 \le c < 40$, is implicitly given for every bucket. With such an identical loose constraint, all the buckets may become candidates.

4.1.2 Traversing Multi-Dimensional Index

Similar to multi-dimensional histogram, multi-dimensional index is also a partition scheme where buckets are specified by intervals over individual attributes. Differently, multi-dimensional index provides access to tuples, but does not contain cardinality information.

Partitioning Space: The index nodes can be viewed as the buckets. There exists a hierarchy in the index tree, thus a hierarchy for the buckets as well. Common multi-dimensional index can be tuple-partitioning (*e.g.*, R-tree [18]) where the buckets may overlap although one tuple only belongs to one leaf node, or space-partitioning (*e.g.*, grid file [27]) where the buckets do not overlap. For instance, the partition in Figure 1 can be a grid file.

Computing Cardinalities: To compute the cardinalities of all the buckets, we have to count the tuples in the corresponding index nodes, resulting in a full traversal of the index tree. This cost can be avoided by augmenting each index node with the number of tuples in the node (and the descendant nodes). Such a variant was briefly discussed in [22]. Recording the cardinality takes several extra bytes per node, resulting in smaller number of pointers that can be stored in each node. Specifically, the consequence in R-tree is smaller fan-out of tree nodes, thus deeper index tree [22].

Retrieving Candidates: The leaf index nodes in a multi-dimensional index provide pointers to individual tuples in the corresponding buckets.

In answering inverse ranking queries, multi-dimensional index has some problems similarly existing for multi-dimensional histogram. For instance, the attributes in the index may not match the attributes of the ranking function. Therefore the score bounds can be loose, resulting in a large number of candidate buckets, as we analyzed in Section 4.1.1. The problem of mismatching between indexed attributes and ranking attributes seriously limits the applicability of this method, since it is not affordable to exhaustively build indices for all possible combinations of dimensions.

Moreover, for multi-dimensional index with tuple-partitioning (*e.g.*, R-tree), the boundary of an index node (*e.g.*, minimal bounding rectangle or MBR, in R-tree) is determined towards the efficiency of the insertion/deletion operations over index, which may conflict with the efficiency in answering inverse ranking queries. For instance, the MBR in R-tree may stretch over a big range along one dimension, resulting in a large interval between the upper-bound and lower-bound of the corresponding bucket. The consequence is a large number of candidate buckets.

4.1.3 Intersecting B-tree Indices

While the method in Section 4.1.2 relies on the existence of multi-dimensional index, we can also partition the space by intersecting the indices (such as B-trees) over individual attributes. Such index intersection was used in evaluating selection queries [26].

Partitioning Space: A B-tree index naturally provides a partition of an attribute domain into multiple intervals. Therefore by intersecting multiple B-trees, we obtain a tuple-space partition.

Computing Cardinalities: Index intersection supplies the list of tuple pointers in each bucket, and thus the cardinality.

Retrieving Candidates: Again, the results of index intersection provide the (pointers to) tuples in each bucket.

The advantage of using index intersection is that we can handle any combination of ranking attributes as long as the individual indices are available. However, unlike multi-dimensional index where the space is readily partitioned, index intersection requires explicit operations to partition the space. The main overhead of this method thus lies in the partitioning, where the B-tree over each attribute must be fully traversed, and the tuple lists are intersected. The result of partitioning is the tuple lists for all the buckets. The traversal on each index may repeat multiple times if the memory buffer cannot hold all the index nodes from all the indices.

4.1.4 Intersecting Bitmap Indices

The method proposed in this section intersects bitmap indices instead of B-trees. While the essential methodology is the same for intersecting both types of index, the key is that bitmap index is more efficient to intersect. Below we first give a brief review of bitmap index, then discuss how to use such index to answer inverse ranking queries.

As an efficient index for dealing with complex decision support queries, bitmap index [28, 29, 30] has gained broad interests and has been adopted in commercial systems. For a bitmap index on an attribute, there exists a bitmap (a vector of bits) for each unique attribute value. The length of the vector equals the number of tuples in the indexed relation. With respect to the vector for value x of attribute a, its i^{th} bit is set to 1, when and only when the

TID	B_a^1	B_a^2	B_a^3	B_a^4
r_1	1	0	0	0
r_2	0	0	1	0
r_3	1	0	0	0
r_4	0	1	0	0
r_5	0	0	0	1
r_6	1	0	0	0
r_7	0	0	1	0
r_8	0	0	0	1
r_9	0	0	1	0
r_{10}	0	1	0	0

(a) bitmap index on $R.a$

TID	B_b^1	B_b^2	B_b^3	B_b^4
r_1	0	0	0	1
r_2	0	1	0	0
r_3	0	0	1	0
r_4	0	0	1	0
r_5	1	0	0	0
r_6	0	1	0	0
r_7	0	0	1	0
r_8	0	0	0	1
r_9	0	1	0	0
r_{10}	0	0	1	0

(b) bitmap index on $R.b$

Figure 6: Example of bitmap indices.

value of a on the i^{th} tuple is x, otherwise 0. With bitmap indices, complex selection queries can be efficiently answered by bit-wise logical operations (AND, OR, XOR, and NOT) over the bit vectors. Moreover, as studied in [30], bitmap indices also enable the efficient computation of some common aggregates, such as SUM and COUNT.

The original form of bitmap index has a problem with high-cardinality attributes. One bit vector must be created for each attribute value in the domain, resulting in big overhead of storage and maintenance if an attribute has many values. In tackling this challenge, researchers have studied a variety of ways in encoding bitmap index [8, 35]. For example, binning can be used to merge the bit vectors for a range of attribute values. Moreover, *bit-sliced index* (BSI) [31] directly capture the binary representations of attribute values.

Partitioning Space: Similar to the aforementioned several methods, the buckets are specified by intervals on individual attributes.

Computing Cardinalities: The IDs of tuples within one specific interval on an attribute are given by the corresponding bit vector. Tuples inside a bucket can thus be obtained by intersecting (AND operation) all the relevant bit vectors. A 1 bit in the resulting vector indicates that the corresponding tuple belongs to the bucket. Computing the cardinality of the bucket is thus a COUNT operation on the resulting vector.

Retrieving Candidates: The union (OR operation) of the vectors for all the candidate buckets gives us a single vector, where the 1 bit corresponds to a candidate tuple. Thus we get the IDs of all candidate tuples.

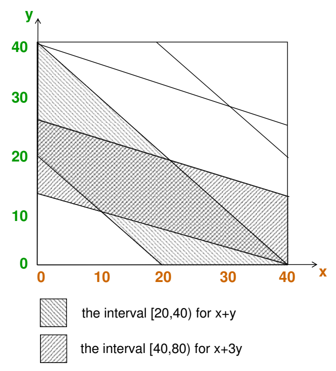
We illustrate the idea using the following example.

Example 8: Figure 6(a) and (b) show the bitmap index on attribute a and b over relation R. The values of a and b are grouped into 4 intervals, respectively, following the partition in Figure 1. Therefore there are 4 bit vectors on a and 4 on b as well. For instance, the 4th bit of vector B_a^2 is 1, which indicates that the value of attribute a for tuple r_4 (*i.e.*, the tuple with ID 4) is in the 2nd interval, *i.e.*, $[10, 20)$. Performing AND operation on each pair of vectors (one for a and another for b), we obtain the resulting vectors for the 16 buckets. For instance, the result of B_a^2 AND B_b^3 is 0001000001, which contains two 1-bits, thus the cardinality of bucket $\{10 \le a < 20,\ 20 \le b < 30\}$ is 2. (Note that we use the same partition as Figure 1, but different cardinalities, for simplicity in illustrating the idea.) ∎

The advantage of using bitmap index is that the bit operations in getting the vectors for buckets are much more efficient than traversing the B-tree nodes and intersecting the verbatim list of tuple IDs. However, in multi-dimensional space with many intervals on each dimension, the number of buckets and thus the number of bitmap operations can be fairly high, resulting in prohibitive cost of computing cardinalities.

TID	x	y
r_1	1	0
r_2	3	3
r_3	2	4
r_4	0	3
r_5	2	1
r_6	4	3
r_7	0	0
r_8	1	3
r_9	3	1
r_{10}	1	4

(a) The relation R.



(b) The intersection of bitmap indices.

TID	$B^{[0,2)}$	$B^{[2,4)}$	$B^{[4,6)}$	$B^{[6,8)}$
r_1	1	0	0	0
r_2	0	0	1	0
r_3	0	0	1	0
r_4	0	1	0	0
r_5	0	1	0	0
r_6	0	0	0	1
r_7	1	0	0	0
r_8	0	1	0	0
r_9	0	1	0	0
r_{10}	0	0	1	0

(c) Bitmap index for $x + y$.

TID	$B^{[0,4)}$	$B^{[4,8)}$	$B^{[8,12)}$	$B^{[12,16)}$
r_1	1	0	0	0
r_2	0	0	1	0
r_3	0	0	0	1
r_4	0	0	1	0
r_5	0	1	0	0
r_6	0	0	0	1
r_7	1	0	0	0
r_8	0	0	1	0
r_9	0	1	0	0
r_{10}	0	0	0	1

(d) Bitmap index for $x + 3y$.

Figure 7: The intersection of bitmap indices on functions.

4.2 Implementations of Partitioning by Function Constraints

4.2.1 *Intersecting Bitmap Indices on Functions*

Following the intuition of using function constraints (Section 3.4), we propose a method of intersecting bitmap indices on functions. Different than the method in Section 4.1.4, the bitmap indices intersected in this approach are built upon ranking functions instead of individual attributes. The motivation in building bitmap index instead of B-tree index on the functions is that, intersecting bitmap index is more efficient than intersecting B-trees.

During index construction, a bitmap index is created for each selected ranking function. It consists of several bit vectors, each of which corresponds to an interval of ranking scores over the ranking function. For a database tuple, its corresponding bit in the vector for the interval subsuming its ranking score is set to 1, and the same bits in other vectors are 0. During query answering, we select one or more indices (functions) that are close to the ranking function in the query. The constraints of the buckets are specified by intervals of scores over the chosen functions. After that, the procedures of computing cardinalities and retrieving candidates are essentially the same as in Section 4.1.4. Note that the ranking functions and the indexed functions must be linear functions in order to make this method applicable.

Example 9: Consider ranking functions $w_1 \times\ a_1 + ... + w_n\ \times a_n$, where each a_i is an attribute and w_i is the corresponding weight. Given a specific combination of $(w_1, ..., w_n)$, the tuples in the space are ranked in the order of the contour lines $w_1 \times a_1 + ... + w_n \times a_n = s$, where s is the ranking score. Therefore we can construct a bitmap index for the given function, *i.e.*, the $(w_1, ..., w_n)$, with several bitmaps. Each bitmap of the index corresponds to a score interval.

Figure 7(a) shows a relation R with its tuples and the attribute values. Suppose the ranking function in the query is x+2y. Among the functions with corresponding bitmap indices constructed, the two functions x+y and x+3y are chosen for answering the query, since they are close to x+2y. Their indices are shown in Figure 7(c) and (d), respectively. Specifically, the index for x+y consists of 4 bitmaps, corresponding to the score intervals $[0, 2)$, $[2, 4)$, $[4, 6)$, and $[6, 8)$. The bitmap index for x+3y consists of 4 bitmaps as well, corresponding to the intervals $[0, 4)$, $[4, 8)$, $[8, 12)$, and $[12, 16)$. The boundaries between these intervals, *i.e.*, the contour lines, are shown in Figure 7(b). The intersections of these intervals give the partition of the tuple space. For instance, the bucket corresponding to the intersection of the two shaded areas has constraints $\{2{\le}x{+}y{<}4, 8{\le}x{+}3y{<}12\}$. The upper-bound and lower-bound scores for this bucket are 6 and 3, respectively, based on linear programming. The bitmap for $2{\le}x{+}y{<}4$ is 0001100110, and the bitmap for $8{\le}x{+}3y{<}12$ is 0101000100. Therefore the bitmap for the shaded bucket is 0001100110 AND 0101000100 = 0001000100 . That is, tuples r_4 and r_8 are in this bucket. ■

4.2.2 Heuristics for Choosing Index to Build

We discuss two index selection heuristics. The first heuristic, *random selection*, is simply to choose arbitrary functions to build index for. Clearly this strategy has the problem of exponential explosion- As the number of attributes involved increases, the hope of an arbitrary indexed function getting close to a future dynamic query is slim. This "curse of dimensionality" is well known in many other areas, such as multi-dimensional indexing.

To address the dimensionality problem, our second heuristic, *workload-based selection*, is to build bitmap indices for those functions that capture the query workload. By doing that, we achieve efficiency for the more frequent queries and sacrifice the less frequent ones. To be more specific, each linear ranking function can be viewed as a point in a multi-dimensional space. Given a set of previous queries, *i.e.*, a set of points in the space, we partition the space into buckets. [10] Associated with each bucket is a *virtual query*, located at the center of that bucket. We thus capture the queries in the bucket as a set of queries identical to the virtual query. This is based on the intuition that if the partition of the query space is fine-grained enough, the queries inside the same bucket are close enough to each other. After measuring the number of queries in each bucket, we choose to build bitmap indices on the virtual query functions of those buckets that contain a large number of queries.

The workload-based selection is effective only when there do exist frequent queries in the workload, *i.e.*, the workload is clustered. In other words, if the queries in the space have equal probability to be issued by users, then the strategy degrades to the above random selection heuristic. For (inverse) ranking queries, it is natural that the workload is clustered. *First*, in ranking a certain type of objects, there usually exists "common sense". *Second*, even though different people have different ranking criteria, similar users share common interests.

4.2.3 Heuristics for Choosing Index to Intersect

With the bitmap indices built for the workload, given a new query, we select two indices that are closest to the query and intersect them. To be more specific, suppose the linear ranking function in the query is f: $w_1\times a_1+...+w_n \times a_n$, where w_i is the weight and a_i is the attribute. Given an indexed function g: $w'_1\times a'_1+...+w'_n \times a'_n$, the closeness of f and g is defined as their cosine similarity,

$$closeness(f, g) = \frac{\overrightarrow{v_f} \cdot \overrightarrow{v_g}}{\|\overrightarrow{v_f}\| \, \|\overrightarrow{v_g}\|}, \qquad (3)$$

[10]The space and buckets of queries should not be confused with the space and buckets of tuples in Section 3.1.

where $\overrightarrow{v_f}$=$<w_1, \ldots, w_n>$ and $\overrightarrow{v_g}$=$<w'_1, \ldots, w'_n>$.

Note that although this method is also based on multi-dimensional space, it does not suffer from the attribute mismatching problem for the approach utilizing multi-dimensional index (Section 4.1.2). The reason is that a function such as $w_1\times a_1$ is a special case of functions involving more attributes such as $w_1\times a_1+w_2\times a_2$. Therefore as long as $w_1\times a_1$ appears frequently, the workload on dimensions (a_1,a_2) is able to capture it.

4.3 Dealing with Boolean Conditions

Up to this point, we always assume the nonexistence of Boolean conditions, *i.e.*, we assume the context relation $R_{\mathcal{B}}$ is simply one base table. This assumption was made only for the purpose of easy presentation. In fact, logical bit vector operations easily allow us to integrate the techniques in Section 4.2.1 with Boolean conditions. Before performing the intersections of bit vectors as shown in Figure 7, the vectors over the indexed function intervals must reflect the filtering effects of the Boolean conditions. If a tuple does not belong to the context relation $R_{\mathcal{B}}$, we must set its corresponding bits in the vectors to 0. The details are further explained below. Note that the techniques applied here are similar to those in [24], which uses bitmap index to incorporate clustering with ranking, together with Boolean conditions.

Selections: Suppose our query has a set of conjunctive range selection conditions $l_1{\le}c_1{\le}u_1, \ldots, l_j{\le}c_j{\le}\ u_j$. We first obtain a vector $vec_{\mathcal{B}}$, where the corresponding bits for the satisfying tuples in $R_{\mathcal{B}}$ are all set. Then given the bit vectors of the indexed functions to be intersected (such as the ones in Figure 7(c) and (d)), we intersect $vec_{\mathcal{B}}$ with each vector, to derive the filtered vectors.

There are many works in the literature on answering Boolean queries using bitmap index. The essential idea is, to get $vec_{\mathcal{B}}$, we compute one vector vec_{c_i} for each condition $l_i{\le}c_i{\le}u_i$, such that vec_{c_i} contains the bits for tuples satisfying the condition. After intersecting the vectors of all the conditions, the resulting vector is $vec_{\mathcal{B}}$. The bit vector vec_{c_i} is computed using bitmap operations over the bitmap index on c_i. In order to get vec_{c_i} with a small number of bitmap operations, it may be necessary to build the bitmap index using some encoding schemes (*e.g.*, [8, 35, 34]). For instance, some scheme requires only one bitmap operation for any one-side range selection condition and some requires only two operations for any two-side condition.

Joins: When join conditions exist in queries, we assume the tables have a snowflake-schema, consisting of one fact table and multiple dimension tables. There are multiple dimensions, each of which is described by a hierarchy, with one dimension table for each node on the hierarchy. The fact table is connected to the dimensions by foreign keys. The tables on each dimension are also connected by keys and foreign keys. As a special case of snowflake-schema, star-schema has only one table on every dimension, thus no hierarchy.

Under such assumption, our technique can be easily extended to handle join queries. Such join queries are so-called "star-joins" under star-schema. Consider a simple case with only two tables, where S is the fact table and R is the dimension table, $j1$ is a key of R and $j2$ is the corresponding foreign key in S. Due to the foreign key constraint, there exists one and only one tuple in R joining with each and every tuple $s \in S$. Therefore for a join condition $R.j1$=$S.j2$, virtually all the join results are in S, with some attributes stored in S and some other attributes in R. Therefore, for each attribute a in the schema of R except $j1$ (since $R.j1$=$S.j2$ and we already have $j2$ in S), we can construct a bitmap index on a for the tuples in S, even though a is not an attribute of S. In general, we can follow this way to construct bitmap index for the tuples in the single fact table, on all relevant attributes in the dimension tables. Thus the Boolean selection conditions involving these attributes can be viewed as being applied on the fact table only. A join query can then be processed like a single table query. More details about such idea of building bitmap join index are in [29].

5. EXPERIMENTS

The algorithms are implemented in C++. The bitmap index is based on [32], which builds multiple bitmap indices at different domain resolutions and compresses them using the WAH compression method [34]. The B-tree index intersection algorithm is built upon an publicly available B-tree implementation in *libgist*, an library that implements GiST [19].

We conducted experiments to compare the various implementation methods in Section 4, together with the straightforward exhaustive approach. Moreover, we investigated how the algorithms are affected by important factors under various configurations. The compared algorithms are: exhaustive method using sequential scan on single table (*Scan*), exhaustive method using sort-merge join (*SMJ*), B-tree intersection (*Btree*), intersecting bitmap index on attributes (*BAttr*), intersecting bitmap index on randomly selected functions (*BFuncRND*), and intersecting bitmap index on workload-based functions (*BFuncWKLD*).

5.1 Experimental Settings

Our experiments were conducted over synthetic tables. The schema of a table consists of a set of attributes, altogether with the total length of 100-byte. Some of the attributes are 4-byte floating number ranking attributes. The values of the ranking attributes are independently generated by various distributions, including uniform, Gaussian, and cosine distributions. Each query used in our experiments uses the weighted sum of the ranking attributes as the ranking function.

We also experiment with join queries in star-schema. The join condition is $A.j$=$B.j1$ AND $B.j2$=$C.j$, where $A.j$ and $C.j$ are keys of A and C, respectively. $B.j1$ and $B.j2$ are the foreign keys in B referring them. A, B, and C have the same size. Among the tuples in A, about a half do not join with any tuple in B. Each tuple in the rest half in average joins with 2 tuples in B. The same statements apply to the join between C and B.

The ranking functions in the queries are weighted-sum functions. We experiment with various number of attributes involved in the functions. The workload is created by a data generator for clustering algorithms from [16]. Viewing each query ranking function as a point, *i.e.*, a vector of weights, in the query space, the workload is a set of clusters. The generator creates values based on underlying data models, one model per cluster. A model specifies, for the corresponding cluster, the mean and standard deviation of each weight individually. The values for a weight are generated by Gaussian distribution with the specified mean and standard deviation.

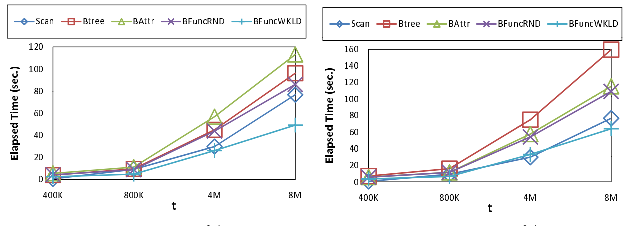
The experiments were conducted on a PC with 2.8GHz Intel Xeon SMP (dual hyperthreaded CPUs each with 1MB cache), 2GB RAM, and a RAID5 array of 3 146GB SCSI disks, running Linux 2.6.15.

5.2 Experimental Results

We evaluated the performances of various methods and studied how they are affected by several important configuration parameters, which are summarized in Table 1. For *BFuncRND* and *BFuncWKLD*, by default there are bitmap indices built for 200 functions. For *BFuncWKLD*, the functions are chosen based on a query workload containing 500 queries. For each index on one function, we use the BSI [31] mentioned in Section 4.1.4. To be more specific, the tuples' values on a function are partitioned into multiple ranges. The binary representation of these range numbers on this function is kept in v (Table 1) vectors, which can 2^v ranges.

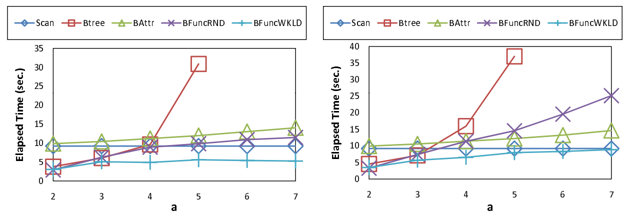
parameter	meaning	values
t	# tuples	400K,800K, 4M, 8M
a	# ranking attributes	2,3,4,5,6,7
q	rank of the query tuple (in percentage)	1%, 10%, 25%, 50%
i	# index built	100, 200, 300, 400
v	# vector per index	7, 8, 9, 10

Table 1: Configuration Parameters.



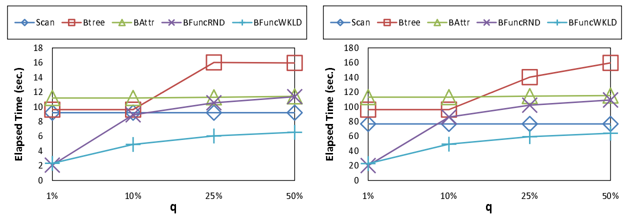
(a) a=4, q=10%. (b) a=4, q=50%.

Figure 8: Single table queries: Execution time varying by t.



(a) t=800K, q=10%. (b) t=800K, q=50%.

Figure 9: Single table queries: Execution time varying by a.



(a) t=800K, a=4%. (b) t=8M, a=4%.

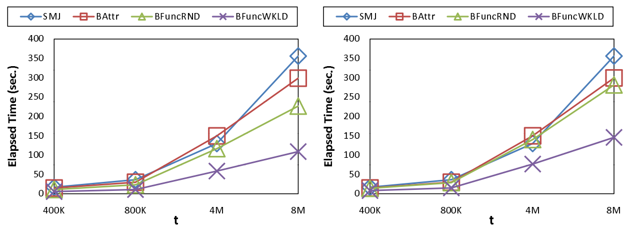
Figure 10: Single table queries: Execution time varying by q.

Single-Table Queries:
To evaluate the performance of single-table queries, we conducted experiments under groups of configurations by the value combinations of the three relevant parameters, t, a, and q. In each group of experiments, we varied the value of one parameter and fixed the values of the rest two. We then run all the algorithms and studied how their performances are affected as the value of the varying parameter changed. The results on wall-clock execution time under six sample groups of experiments are shown in Figure 8, 9, and 10. From the figures, we can make the following observations:

First, *BFuncWKLD* is usually the best algorithm and it is several times more efficient than others. This validates the approach of using bitmap index built upon query workload.

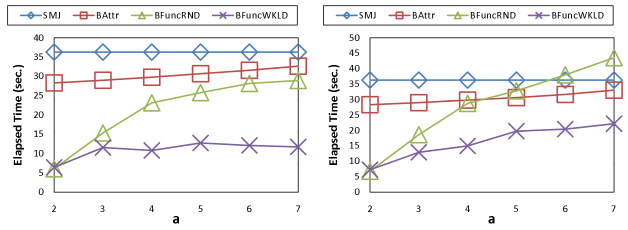
Second, for single-table queries, *Scan* performed pretty well in comparison with various other methods, except *BFuncWKLD*. This observation verifies our analysis of the various methods in Section 4.1. They all have significant disadvantages. For instance, Figure 9 shows that, as the number of ranking attributes increases, the performance of intersecting B-trees degrades exponentially due to the fact that it has to fully traverse all the B-trees and intersect the tuple pointers. As another example, Figure 9(b) clearly illustrates the "curse of dimensionality" on using bitmap indices built upon randomly selected functions, as mentioned in Section 4.2.3.

Third, from Figure 10 we can see that the rank position is also important in determining the efficiency. number of rank position. Especially, the smaller rank position, the more efficient *BFuncRND* and *BFuncWKLD* are. To obtain the rank of an object that is ranked at 1% (*e.g.*, the object ranked at 8192 when t=800K), *BFuncRND* outperforms *Scan*. However, as the rank position increases, it becomes worse than *Scan*. This figure clearly shows that inverse ranking queries are more challenging than *top-k* queries, in the sense that efficient approach for obtaining objects at small k may become even worse than the straightforward approach.



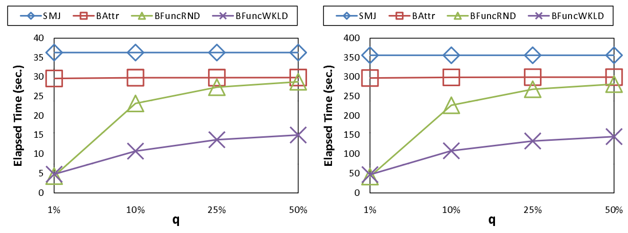
(a) a=4, q=10%. (b) a=4, q=50%.

Figure 11: Join queries: Execution time varying by t.



(a) t=800K, q=10%. (b) t=800K, q=50%.

Figure 12: Join queries: Execution time varying by a.



(a) t=800K, a=4%. (b) t=8M, a=4%.

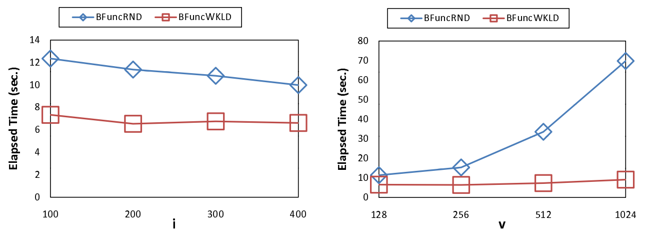
Figure 13: Join queries: Execution time varying by q.

Join Queries:
The results of join queries are shown in Figure 11- 13, in the same fashion as the results for single-table queries. Note that *SMJ* replaces *Scan* for joining, and *Btree* becomes unapplicable for join queries. From the figures, we can make the following observations:

First, *BFuncWKLD* is still clearly the best algorithm, and its performance advantages over other algorithms are enlarged under join.

Second, different from the single-table scenario, the exhaustive approach *SMJ* now is often the worst method. This is due to the fact that a full join will scan large number of tuples and join large number of intermediate results. While other methods are able to zoom into candidate tuples, thus reduce the number of I/Os.

Third, *BFuncRND* is often the second best method under small number of ranking attributes. However, the "curse of dimensionality" is illustrated again by Figure 12.



(a) t=800K, a=4, q=50%, varying i. (b) t=800K, a=4, q=50%, varying v.

Figure 14: Single table queries: Execution time varying by i and v.

To further understand *BFuncRND* and *BFuncWKLD*, we analyze how the parameters affect their performances, as shown in Figure 14. As expected, as the number of built indices increases (Figure 14(a)), the algorithms are more efficient. However, 400 indices do not give us significant performance improvements over 100 indices. This indicates that the small number of indices are sufficient for the given workload. Under other workload, more indices may be required. In Figure 14(b), we see that increasing the number of vectors in fact makes the performance of *BFuncRND* worse. The reason is that randomly selected functions cannot match the query functions well, resulting in a large number of candidate buckets. As there are more vectors per index, the partition has more buckets, therefore *BFuncRND* needs to intersect more bit vectors, compute the cardinalities for more candidate buckets, resulting in degraded performance. On the other hand, *BFuncWKLD* is not seriously affected by v, indicating that the workload-based functions can successfully capture the queries, resulting in small number of intersections and candidate buckets.

6. RELATED WORK

To the best of our knowledge, this is the first piece of work that investigates inverse ranking queries. In spite of the semantic connections between ranking and inverse ranking, techniques developed for top-k queries [15, 7, 14, 9, 2, 21, 10, 23, 20, 33, 13] cannot help in dealing with inverse ranking queries. As discussed in Section 1, top-k algorithms are optimized for small k, instead of arbitrary ranking positions. As k increases, their performances degrade and eventually become even worse than the materialize-then-sort approach. Specifically, in [9], top-k selection queries are transformed into range queries, based on multi-dimensional histograms. Note that it is inefficient to use similar approach to process inverse ranking queries, as Section 4.1.1 indicated.

[20] studies using materialized ranked results to answer ranking queries. [13] presents a more general approach of using views in answering *top-k* queries. It also uses linear programming during query processing. However, it focuses on small number of top k answers, and thus it is less important to build efficient access method (such as the index) for retrieving previous query results.

This is also the first work to study quantile queries in the general context of querying databases. There were quite some works on computing quantiles in databases and stream data systems [1, 3, 17, 25, 4, 12, 11]. However, they are fundamentally different from the quantile queries in this paper. *First*, they focus on the quantile of a set of data items, where the values of the items themselves give their ranks. To the contrary, this paper studies general quantile queries, where database records are ranked by ranking functions that involve multiple attributes or even multiple tables. *Second*, our quantile queries are in the context of general database queries, whereas they do not consider the integration with Boolean query conditions. *Finally*, from the perspective of application domains, the quantile queries in this paper are for flexible data retrieval, exploration, and analysis. The previous works do not consider such applications. They use quantiles of data for query optimization, result size estimation, association rule mining, data cleaning, and data partitioning.

[36] studies quantile retrieval on multi-dimensional data. However, quantiles in that work are based on a single measure associated with multi-dimensional index structure, instead of ranking function. Therefore it does not incur the problem of mismatching between indexed attributes and ranking attributes. Moreover, it does not consider the integration with Boolean query conditions.

7. CONCLUSION

We propose a new type of inverse ranking queries, that obtain the ranks of given query objects among certain contact objects. Such queries are useful in supporting data retrieval and exploration, thus can be important in many applications. We further develop a framework for processing these queries, and discuss several alternative methods within this general framework. Some of these methods utilize popular data structures existing in current database systems and some others are based on the new data structures proposed in this paper. We analytically study the cost model of these methods, and empirically compare them with each other as well, to understand the tradeoff in applying these methods. We also compare these methods with the traditional straightforward method, and verify that our method is much more efficient. To the best of our knowledge, ours is the first work that studies inverse ranking queries. We believe that we have conducted a thorough investigation on the definition of the novel queries, the design of efficient processing methods for such queries, and the experimental evaluation of the methods.

8. REFERENCES

[1] R. Agrawal and A. Swami. A one-pass space-efficient algorithm for finding quantiles. In *Proc. 7th Int. Conf. Management of Data, COMAD*, 1995.

[2] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. In *CIDR*, 2003.

[3] K. Alsabti, S. Ranka, and V. Singh. A one-pass algorithm for accurately estimating quantiles for disk-resident data. In *VLDB*, pages 346–355, 1997.

[4] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *PODS*, pages 286–296, 2004.

[5] D. Bertsimas. *Nonlinear Programming*. Athena Scientific, Belmont, Massachusetts, 2nd edition, 1995.

[6] D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, Belmont, Massachusetts, 1997.

[7] M. J. Carey and D. Kossmann. On saying "enough already!" in SQL. In *SIGMOD*, 1997.

[8] C. Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *SIGMOD*, 1999.

[9] S. Chaudhuri and L. Gravano. Evaluating top-*k* selection queries. In *VLDB*, pages 397–410, 1999.

[10] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR technologies: What is the sound of one hand clapping? In *CIDR*, pages 1–12, 2005.

[11] G. Cormode, M. N. Garofalakis, S. Muthukrishnan, and R. Rastogi. Holistic aggregates in a networked world: Distributed tracking of approximate quantiles. In *SIGMOD*, pages 25–36, 2005.

[12] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Effective computation of biased quantiles over data streams. In *ICDE*, pages 20–31, 2005.

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

[14] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top n queries. In *VLDB*, 1999.

[15] R. Fagin, A. Lote, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[16] F. Farnstrom, J. Lewis, and C. Elkan. Scalability for clustering algorithms revisited. 2(1):51–57, August 2000.

[17] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *VLDB*, pages 454–465, 2002.

[18] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[19] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *VLDB*, pages 562–573, 1995.

[20] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. *SIGMOD*, 2001.

[21] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join quereis in relational databases. In *VLDB*, pages 754–765, 2003.

[22] Y. E. Ioannidis. The history of histograms (abridged). In *VLDB*, pages 19–30, 2003.

[23] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. RankSQL: Query algebra and optimization for relational top-k queries. In *SIGMOD*, pages 131–142, 2005.

[24] C. Li, M. Wang, L. Lim, H. Wang, and K. C.-C. Chang. Supporting ranking and clustering as generalized order-by and group-by. In *SIGMOD 2007 (to appear)*.

[25] X. Lin, H. Lu, J. Xu, and J. X. Yu. Continuously maintaining quantile summaries of the most recent n elements over a data stream. In *ICDE*, pages 362–374, 2004.

[26] C. Mohan, D. J. Haderle, Y. Wang, and J. M. Cheng. Single table access using multiple indexes: Optimization, execution, and concurrency control techniques. In *EDBT*, pages 29–43, 1990.

[27] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.

[28] P. E. O'Neil. Model 204 architecture and performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 40–59, September 1987.

[29] P. E. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, 1995.

[30] P. E. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD*, pages 38–49, 1997.

[31] D. Rinfret, P. O'Neil, and E. O'Neil. Bit-sliced index arithmetic. In *SIGMOD*, pages 47–57, 2001.

[32] R. R. Sinha, S. Mitra, and M. Winslett. Bitmap indexes for large scientific data sets: A case study. In *IPDPS*, 2006.

[33] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava. Ranked join indices. In *ICDE*, 2003.

[34] K. Wu, E. Otoo, and A. Shoshani. An efficient compression scheme for bitmap indices. 31:1–38, 2006.

[35] M.-C. Wu and A. P. Buchmann. Encoded bitmap indexing for data warehouses. In *ICDE*, pages 220–230, 1998.

[36] M. L. Yiu, N. Mamoulis, and Y. Tao. Efficient quantile retrieval on multi-dimensional data. In *EDBT*, pages 167–185, 2006.