On Contextual Ranking Queries in Databases

Chengkai Li*

Department of Computer Science and Engineering
University of Texas at Arlington
Arlington, TX 76019, USA

Abstract

In this paper, we identify a novel and interesting type of queries, *contextual ranking queries*, which return the ranks of query tuples among some context tuples given in the queries. Contextual ranking queries are useful for OLAP and decision support applications in non-traditional data exploration. They provide a mechanism to quickly identify where tuples stand within the context. In this paper, we extend the SQL language to express contextual ranking queries and propose a general partition-based framework for processing them. In this framework, we use a novel method that utilizes bitmap indices built on ranking functions. This method can efficiently identify a small number of candidate tuples, thus achieves lower cost than alternative methods. We analytically investigate the advantages and drawbacks of these methods, according to a preliminary cost model. Experimental results suggest that the algorithm using bitmap indices on ranking functions can be substantially more efficient than other methods.

Keywords: ranking, bitmap index, decision support

1. Introduction

In OLAP and decision support applications, users often want to express non-traditional database queries with "soft" criteria that capture notions such as similarity, relevance, and preference, in contrast to standard *Boolean queries* (i.e., queries with Boolean selection and join conditions). Proposals such as top-k ranking queries, skyline queries, and preference queries have gained popularity in recent years (e.g., [1, 2, 3, 4, 5]). In this paper, we identify a novel and interesting type of *contextual ranking queries*, symmetrical to ranking. A contextual ranking query obtains the rank of a query tuple within a context given in the query. The rank indicates how many tuples in the context have higher ranking scores than the query tuple.

Contextual ranking queries can be useful in many places. For instance, when selling a product in an online market, the seller may want to know how popular the product would be, in order to set an appropriate price. As another example, we may want to compare a house to others with regard to region, size, year, etc. The applications of contextual ranking could include store location selection, online advertising placement, player drafting in fantasy sports games, and so on. A motivating example is as follows.

Example 1 Consider a home building company that is planning to build a new house with certain properties at certain location. The company wants to compare the house in plan with the existing houses from all realtors in the city, using certain ranking criteria that capture buyers' preferences. The comparison result, more specifically the rank position of the future house, indicates how popular it might be and thus provides assis-

*Corresponding author
Email address: `cli@uta.edu` (Chengkai Li)

tance in revising the building plan.

The key is that the builder is not looking for top houses among existing ones since she is constrained by the available options in building the new one. Instead she wants to know where the new house would stand compared to the existing ones, and to modify their plan accordingly. The ranking criteria are *ad-hoc*, because different decision makers may have different ways of understanding potential buyers and modeling their preferences. Moreover, the decision makers want to focus on certain subsets of the context (i.e., existing houses), e.g., those in a given district or with a specific property. Furthermore, the builder needs to accomplish the same analysis for various alternatives, i.e., multiple choices in building the new house, whenever such an alternative arises. Thus contextual ranking queries may be processed many times, for varying ranking criteria, context, and query object. ■

To process contextual ranking queries, a simple brute-force approach is to fully materialize the results of a Boolean query, i.e., the context of ranking, count the number of tuples with higher ranking scores than a query object, and thus obtain the query object's rank. This approach can be inefficient, since it has to generate the full Boolean results, even for getting the rank of a single tuple. More importantly, the results of exhaustive computation become invalid whenever query conditions change, thus it is only suitable for infrequent or *one-shot* analytical operations. On the contrary, the contextual ranking problem engaging us in this paper is a day-to-day operation, requested by a large number of users, using diverse ranking attributes and functions upon different result sets from varying Boolean conditions. We thus focus on efficiently supporting such *on-the-fly* analysis and integrating with Boolean conditions.

To answer contextual ranking queries, we must locate where the given query tuples stand among the context tuples. The key to an efficient solution lies in avoiding full materialization by the aforementioned exhaustive approach. 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.

Based on the symmetry between top-k queries and contextual ranking, it may appear that various top-k query algorithms(e.g., [1, 2]) can be adopted, since they also often avoid full materialization. The difference on the surface is that contextual ranking looks for the rank of a given tuple, whereas top-k ranking looks for the tuples ranked within top k. Hence, a natural adoption of a top-k algorithm would operate by continuously getting the "next" top tuple, until it reaches a tuple with a score lower than that of the query tuple. When it terminates, the "k" is the query tuple's rank. However, top-k algorithms are explicitly optimized for retrieving few (k) top answers. When k is relatively large, the performance of these algorithms become even worse than an exhaustive approach [2]. Unfortunately, a contextual ranking query may ask for the rank of a tuple that may be ranked anywhere, corresponding to arbitrary k. Therefore top-k algorithms may only be efficient for contextual ranking queries in special cases (very small k).

We design a general *partition-and-prune* framework for processing contextual ranking queries. It 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 the candidate buckets that consist of tuples ranked near a query tuple. After their cardinalities (number of tuples) are computed, the non-candidate buckets can be safely pruned because their tuples are all ranked higher (lower) than the query tuple. Therefore we only need to retrieve the tuples in candidate buckets, for obtaining the query tuple's rank position. Unsurprisingly such a partition-and-prune (or similarly branch-and-bound) approach has been shared by many works on ranking and top-k queries [2]. However, we emphasize that branch-and-bound is only a high-level paradigm (which is applied in many places including B-tree and R-tree), while the challenges remain in forming the solution to a specific problem under this paradigm.

The method of realizing the framework in an efficient manner does not come straightforwardly.

At the first glance, one may resort to various existing data structures in a DBMS, including multi-dimensional histogram and multi-dimensional index, as they naturally provide a *partition* of tuples. However we note that, without going into the technical details, the practicality of these choices in answering contextual ranking queries is seriously limited. They could only be competitive in special cases. *First*, multi-dimensional histograms or indices must be constructed beforehand for a given set of attributes. When ranking attributes do not match the index/histogram attributes, they can only obtain loose upper and lower bounds, resulting in a prohibitively large number of candidate buckets, thus losing the effect of *pruning*. Unfortunately in reality it is not affordable to exhaustively build histograms or indices for all possible combinations of ranking attributes. Therefore the "curse of dimensionality" significantly limits the applicability of such methods. *Moreover*, there is little room to further allow Boolean conditions. Although methods exist for building and utilizing multi-dimensional histograms and indices for joins, the combination of ranking and Boolean attributes from joined tables will only increase the number of dimensions that need to be matched by these structures, resulting in a heavier curse of dimensionality. An alternative to multi-dimensional structures is to partition by intersecting single-dimensional structure such as B-trees. However, such a method incurs large overhead due to the overwhelming traversal of tree nodes and intersections of their tuple lists. Detailed analysis of these methods are further given in Section 5.2.

Our solution hinges on the idea of a novel multi-dimensional index structure that aligns with ranking functions (Section 5.1). *First*, this index does not partition a tuple space horizontally and vertically, unlike conventional multi-dimensional index. Instead, it partitions by ranges of scores based on a ranking function. By such an alignment, the index generates a small number of candidate buckets, avoiding heavy access to candidate tuples, a problem suffered by conventional index in answering contextual ranking queries. *Second*, while it cannot afford building indices for every possible ranking functions, this method chooses to create indices for functions that are similar to the ones in many previous queries, guided by query workloads. When a future query arrives, it can select indices "close" to the query, with the reasonable expectation that future queries share similar characteristics with the workload and thus the index can potentially match the ranking functions of many future queries. *Finally*, we use bitmap index to implement such function-based index because it allows fast bit operations, including intersecting indices, counting bucket size, and so on.

Experiments (Section 6) show that the method of using bitmap index on ranking functions can significantly outperform alternative approaches. The algorithm achieves good efficiency without requiring too many queries in the workload or too many indices, it maintains efficiency up to a sufficient number of ranking attributes for typical ranking applications, it scales up to large tables, and it allows joins.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 introduces the SQL extension to define contextual ranking queries. Section 4 presents the partition-and-prune framework. Section 5 introduces the method of using function-based bitmap index and other alternatives. We experimentally evaluate the algorithms in Section 6 and conclude in Section 7.

2. Related Work

There are many previous works (e.g., [6, 7, 8]) on computing quantiles on a set of data values (e.g., the 40% quantile of a set $\{10, 15, 4, 23, 11\}$). The q quantile is the value v_q such that a fraction q of the data values are higher (lower) than v_q. In contextual ranking queries, database tuples are ranked by functions combining multiple attributes. In order to apply any such quantile computing algorithm in solving contextual ranking queries, we must at least fully materialize the context tuples and then compute their ranking scores, in order to obtain a set of data values that these algorithms work on. Such full materializa-

tion is exactly what we want to avoid for better efficiency, as mentioned before.

There also exists a duality between contextual ranking queries and *quantile queries* [9]. While a contextual ranking query looks for the rank of a query tuple with known features, a quantile query identifies the features possessed by a tuple at certain rank position. [9] considers using R-tree in processing quantile queries in multi-dimensional space. An R-tree must be constructed beforehand and the dimensions of the R-tree must exactly match the ranking attributes in a quantile query. Although in principle R-tree can be used for computing contextual ranking as well, the requirement of exact attribute match could significantly limit the applicability of this method, when there is mismatch between indexed attributes and ranking attributes. (Detailed analysis in Section 5.2.) Moreover, contextual ranking queries have Boolean conditions (selections and joins) in addition to ranking. Integration with Boolean conditions was not considered by previous works on quantile computing.

The preliminary version of this work appears in [10], where the concept of contextual ranking query was originally proposed, under the term *inverse ranking query*. Lian et al. [11] studied probabilistic inverse ranking queries in uncertain databases. While we focus on precise data and exact answers, their work is on answering queries in uncertain databases with confidence guarantees. One main focus of their work is to derive the probabilistic score bounds. Another focus is on uncertain query objects. Bernecker et al. [12] further studied the problem in the setting of uncertain data streams. Given the very different problem settings and focuses, the techniques developed in our work and [11, 12] are not applicable for each other. These works all follow the general methodology of pruning by upper/lower-bound scores, which is commonly exploited in ranked query processing.

Wan et al. [13] investigated the problem of creating new combinations of products that are not dominated by existing combinations. The dominance relationship between products is based on the concept of skyline [4], in which multiple criteria are not combined by a ranking function.

3. Defining Contextual Ranking Queries

In this section, we propose extensions to SQL language for expressing contextual ranking queries. While syntax is not our focus, a formal description facilitates the discussion of our concept and solution in following sections.

To specify a contextual ranking query, we overload the OLAP function *rank*() in SQL, as follows:

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 a contextual 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* $R_{\mathcal{B}'}{=}\sigma_{\mathcal{B}'(c'_1,\ldots,c'_l)}$ $(R'_1 \times \cdots \times R'_h)$. Following *rank*() **IN**, another Boolean function $\mathcal{B}$ $(c_1, \ldots, c_j)$ over $R_1 \times \ldots \times R_n$ supplies the *context tuples* or *context relation* $R_{\mathcal{B}}{=}\sigma_{\mathcal{B}(c_1,\ldots,c_j)}$ $(R_1 \times \cdots \times R_n)$. A *ranking function* $\mathcal{F}$ over the *ranking attributes* $p_1, \ldots, p_m$ (e.g., $\mathcal{F}{=}p_1{+}p_2{+}p_3$) gives a *ranking score* $\mathcal{F}[t]$ for each context and query tuple t. (Note that the context and query tuples should have the ranking attributes, i.e., the schemata of both $R_{\mathcal{B}}$ and $R_{\mathcal{B}'}$ contain attributes $p_1, \ldots, p_m$.) Formally, Q returns $R_{\mathcal{B}'}$ together with their ranks among $R_{\mathcal{B}}$, by the descending order of their scores. [1] 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., $rank(t_q){=}1{+}|\{t_c|\mathcal{F}[t_c]{>}\mathcal{F}[t_q], t_c{\in}R_{\mathcal{B}}\}|$. [2]

The aforementioned query obtains query tuples $R_{\mathcal{B}'}$ by Boolean selection and join conditions. However, we may be interested in the ranks of virtual tuples that do not necessarily exist. Hence we propose the following alternative syntax that requests the rank of a virtual tuple $p_1{=}v_1, \ldots, p_m{=}v_m$.

[1] We assume **order by asc|desc** uses descending (**desc**) by default.

[2] When there are ties in scores, an arbitrary *deterministic* "tie-breaker" function can determine an order, e.g., by tuple IDs.

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 2 Consider a home builder. To decide how popular an existing house (with id 1001) is in the market, the following query gets the house's rank among those in the same area. Various ranking functions can be used in computing ranking scores. For simplicity, a weighted sum (with both positive and negative weights) function on house price and size is used.

select *hid*, *rank*() **in** (**select** *
from *Houses*
where *zipcode*=12345)
from *Houses*
where *hid*=1001
order by $size - 0.01 \times price$

Alternatively, we might want to use the following query to determine the rank of a (virtual) house that is in plan.

select *rank*() **in** (**select** *
from *Houses*
where *zipcode*=12345)
values (*size*=4000, *price*=250000)
order by $size - 0.01 \times price$ ■

We do not discuss how to obtain $R_{\mathcal{B}'}$. There can be situations when the cost of obtaining $R_{\mathcal{B}'}$ themselves dominates that of getting their ranks. However, we speculate that contextual ranking queries are mostly on small number of query tuples that can be quickly identified, such as by tuple IDs or by fast index on their properties.

4. Partition-and-Prune Framework

This section presents a general framework for processing contextual ranking queries. The framework is simple and intuitive. We partition the space of tuples, i.e., the context relation, into buckets and compute the upper-bound and lower-bound of ranking scores of the tuples within each bucket. The bounds classify the buckets into three

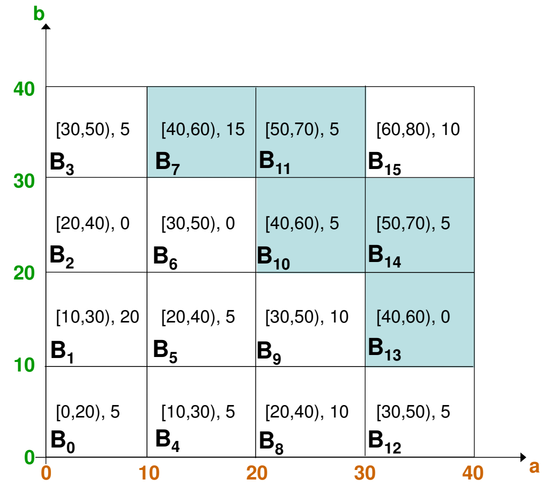


Figure 1: Buckets in a 2-dimensional space.

categories, with regard to a given query tuple t_q. The buckets with lower-bounds higher than $\mathcal{F}[t_q]$ contain context tuples ranked higher than t_q; the buckets with upper-bounds lower than $\mathcal{F}[t_q]$ contain lower-ranked context tuples; and the context tuples in remaining buckets, the *candidate tuples*, may be ranked higher or lower than t_q. Therefore by counting the cardinalities (number of tuples) of buckets, we know how many context tuples are guaranteed to be ranked higher (lower) than t_q, and we only need to look up the scores of candidate tuples, in order to obtain t_q's rank. We illustrate the idea in Example 3, as our running example.

Example 3 Consider a query that 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 tuple space partitioned into 16 buckets on a and b, along with the ranges of a and b for each bucket. For instance, the bucket B_0 has ranges $0 \leq a < 10$ and $0 \leq b < 10$. These ranges determine the upper- 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 with lower- and upper-bound

score 0 and 20, respectively. Among these buckets, the shaded ones are candidate buckets and others are pruned. Bucket B_{15} is pruned because the scores of tuples inside it are at least 60, which is higher than the query tuples' score, 35+20=55. The 10 left lower buckets are also pruned because their tuple scores are below 55. The tuples in candidate buckets may score above or below 55. There are 10 tuples in B_{15} and totally 30 tuples in all candidate buckets. Hence the query tuples' ranks are between 11 and 40. To get their ranks, we obtain the tuples in candidate buckets and resolve their orders to the query tuples. ∎

Below we formally present the framework. Section 4.1 defines the partitioning and pruning of tuple space, as the basis of a general algorithm in Section 4.2. Section 4.3 discusses a cost model, which helps us in designing and analyzing various schemes of partitioning in Section 4.4, and is the guideline in forming our algorithm in Section 5. Throughout the discussion, we assume the context relation $R_{\mathcal{B}}$ is simply one base table, without Boolean conditions over context tuples. In Section 5.1.4, we briefly discuss how to extend the techniques to handle join and selection conditions.

4.1. Tuple Space Partitioning and Pruning

Definition 1 (Partition, Bucket, Constraint) A relation R is a set of tuples $\{t_1, t_2, \ldots\}$ with schema $A=\{a_1, a_2, \ldots\}$. A *partition* $\mathcal{P}_{\mathcal{R}}=\{b_1, b_2, \ldots\}$ is a set of mutual-exclusive subsets of R that cover R, i.e, $\cup b_i=R$, $b_i\neq\emptyset$, and $b_i\cap b_j=\emptyset$, $\forall b_i, b_j\in\mathcal{P}_{\mathcal{R}}$. Each subset b_i is a *bucket*. Given 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 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, consisting of 16 buckets. The cardinalities of the buckets are shown. 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 a 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 the tuples in b, i.e., $\lceil b\rceil>\mathcal{F}[t]$, $\forall t \in b$. Similarly, a lower-bound score $\lfloor b\rfloor$ is a value that is smaller than or equal to the lowest score among the tuples in b, i.e., $\lfloor b\rfloor\leq\mathcal{F}[t]$, $\forall t \in b$. [3] [4] ∎

Definition 3 (Pruned and Candidate buckets) With regard 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 regard 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 $\lfloor B_0\rfloor=0$.[5] Similarly, we obtain the bounds of other buckets. The query tuple has score 55. Hence the white buckets in Figure 1 are pruned and the shaded ones are candidates, by Definition 3. ∎

In determining the query tuple's rank, we can safely discard the pruned buckets and only look

---

[3] 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.

[4] There is an infinite number of upper- and lower-bounds for any bucket.

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

up the candidate tuples, as already illustrated in Example 3. More formally, we have the following property. 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 its rank in $R \cup \{t_q\}$, i.e., $rank(t_q) = 1 + \sum_{b \in pruned^+(\mathcal{P}_{\mathcal{R}}, t_q)} |b| + |\{t_c | \mathcal{F}[t_c] > \mathcal{F}[t_q] \wedge t_c \in b$ $s.t.\ b \in candidate\ (\mathcal{P}_{\mathcal{R}}, t_q)\}|$. ■

4.2. General Algorithm

Based on Property 1, a general algorithm is outlined in Figure 2. It takes the following steps in sequence:

1. Partitioning Space: The algorithm partitions the tuple space into buckets by deciding the number of buckets and their associated constraints. The constraints directly determine the geometrical shape and area of a bucket.

2. Deriving Bounds: The algorithm derives the upper- 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 the general NLP problem is hard, there are methods for special cases [14].

We concentrate on *monotonic* ranking functions, which are commonly studied in top-k queries. Examples of such functions include *sum*, *weighted average*, and *Lp-norm distance* (e.g., *Manhattan* and *Euclidean distance*). With single-attribute constraints in the form of $l \leq a < u$, the bounds of such a monotonic function can be straightforwardly determined by the ranges on attributes. Therefore given a partition by only single-attribute constraints, the algorithm can handle any monotonic ranking function.

Deriving the bounds becomes a *linearly constrained optimization* problem when all constraints are linear functions over the attributes, and further a *linear programming* (LP) problem when the ranking function is a linear function as well. There are well-studied algorithms for solving LP problems, such as the Simplex method [15]. Therefore given a partitioning scheme using linear constraints, the algorithm can process linear ranking functions, a subset of monotonic functions [6] [7].

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

4. Classifying Buckets: The bounds and cardinalities help to identify pruned and candidate buckets, following Definition 3.

5. Retrieving Candidates: It retrieves candidate tuples, evaluates their scores, and obtains the ranks of query tuples.

Among the five steps, step 2 and 4 are shown in Figure 2 and 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 are applicable. An appropriate partitioning scheme is thus key to the efficiency of this approach. In Section 4.3 we describe an analytical cost model of the algorithm, which guides the design of partitioning schemes in Section 4.4.

4.3. Cost Model

The primitive cost model in this section is for analyzing and comparing the choices in realizing our framework [8]. It has the following components:

Cost factors: The cost of our algorithm is determined by several factors, including tuple space partition, data distribution, data size, and query.

Cost parameters: The cost is a function of several parameters, including number of buckets ($|\mathcal{P}_{\mathcal{R}}|$), their score bounds ($\lfloor b \rfloor$ and $\lceil b \rceil$) and cardinalities ($|b|$), and candidate buckets ($candidate(\mathcal{P}_{\mathcal{R}}, t_q)$).

[6] Single-attribute constraint is an extreme case of linear constraint.

[7] A linear function such as $2p_1 - 3p_2$ is monotonic on p_1 and $-p_2$.

[8] A complete cost model in query optimizer is beyond our focus.

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: $pruned^-(\mathcal{P}_{\mathcal{R}}, t_q) \leftarrow \emptyset$ /* dominated buckets */
19: $pruned^+(\mathcal{P}_{\mathcal{R}}, t_q) \leftarrow \emptyset$ /* dominating buckets */
20: $candidate(\mathcal{P}_{\mathcal{R}}, t_q) \leftarrow \emptyset$ /* 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 algorithm outline.

These parameters are determined by the aforementioned cost factors. Specifically, the partition determines the number of buckets and their bounds (constraints), the data distribution and size determine the cardinalities, and the query determines the candidate buckets, together with partition, data distribution and size.

Cost formula: The cost formula in terms of time is the sum of CPU cost, I/O cost for obtaining cardinalities (step 3 in Section 4.2), and I/O cost for retrieving candidate tuples (step 5). Among the five steps in Figure 2, step 1, 2, and 4 do not

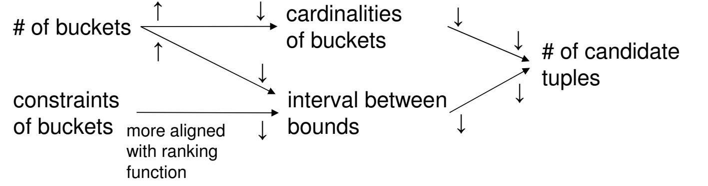


Figure 3: The relationships among cost parameters.

involve disk I/O.

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

Conventional database query algorithms are I/O-bound rather than CPU-bound. Therefore the common practice in investigating query plan cost is to focus on disk I/O cost. However, we shall see that some of the methods in Section 5 involve CPU costs that cannot be ignored.

To achieve the goal of avoiding full materialization of context tuples, we aim to compute cardinalities of buckets (step 3) without fetching their individual tuples. Without getting into the details yet, we note that we intersect bitmap vectors and use COUNT operations on resulting vectors to obtain bucket cardinalities. On the contrary, step 5 needs to retrieve individual tuples in candidate buckets, in order to evaluate their exact scores. The size of a bitmap vector is much smaller than that of the tuples represented by the vector. Therefore, we focus on the disk I/O cost of step 5, for which a good metric is the number of candidate tuples, i.e.,

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

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 these cost parameters, in determining the number of candidate tuples. The up-arrow and down-arrow represent "increase in amount" and "decrease in amount", respectively. *First*, the number of candidate tuples is directly determined by the bounds and cardinalities of buckets. The more tuples in each candidate bucket, the more candidate tuples; and the bigger interval between the

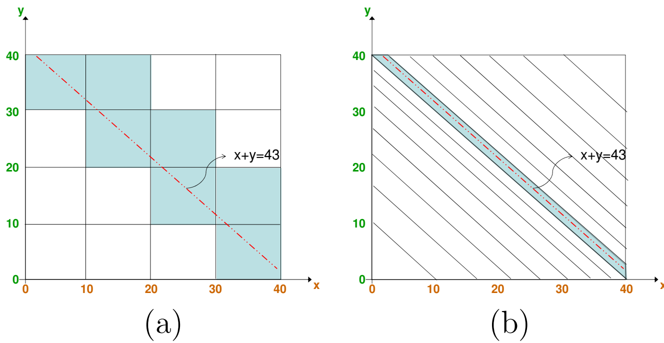


Figure 4: Constraints and bounds.

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 their constraints. 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. It seems to indicate we should have as many buckets as possible, in order to get less candidate tuples. However, with more buckets, the cost of constructing them and computing cardinalities can be significant.

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

Example 6 Figure 4 shows a space over x and y of relation R. We look for the rank of a tuple with score 43, under ranking function $x+y$. Tuples with same scores are on distinct contour lines, one for each score value. For instance, the dashed line in Figure 4 is the contour line of $x+y=43$.

Different constraints can result in very different bounds. Figure 4 shows two partitions of the same space, where the solid lines are boundaries of buckets. The partition in Figure 4(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 $\{l \leq x+y < u\}$, i.e., the constraints are parallel to the contour lines of the ranking function. Both (a) and (b) partition the space into 16 equi-area buckets. However, the buckets in (b) have much smaller intervals between bounds than the buckets in (a) do. Therefore there are 7 candidate buckets (in shade) in (a), while only 1 in (b). In fact, the candidate buckets in (b) are subsumed by the candidates in (a). ■

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.

4.4. Partitioning Schemes

The above analysis indicates that the most significant cost component, the number of candidate tuples, is determined by partitioning scheme, which consists of number of buckets and constraints, by definition. Constraints specify the way of partition, while number of buckets controls the granularity of partition. Below we present several partitioning schemes, i.e., various types of constraints. Their implementations are in Section 5.

Single-Attribute Constraints: A straightforward approach of partitioning is to use 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. The boundaries between buckets are parallel to the dimensions, i.e., attributes. This scheme can support any monotonic functions on these attributes. Moreover, it is easy to conduct satisfaction test for constraints. For instance, a B-tree on a may be used in obtaining those tuples satisfying constraint $l \leq a < u$.

Function Constraints: Partitioning by single-attribute constraints can be sub-optimal. As discussed in Section 4.3, 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 that uses functions as constraints. In this scheme, each constraint has 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. One heuristic to measure closeness is to use 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. Computing score bounds under such constraints is a linear programming problem, as Section 4.2 states. We build bitmap index over g (instead of single attribute a) to perform satisfaction test for function constraints.

Workload-Based Function Constraints: For the above scheme to achieve a small number of candidate tuples, g should be close to f. In other words, we must build indices for various g, so that among them we can find one close to the dynamic f in query. However, as the number of attributes involved in ranking functions increases, the necessary number of indices for g may become prohibitively large. Our idea in tackling this challenge is to 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. With the reasonable expectation that future queries share similar characteristics with the workload, the indexed functions can capture many future queries.

5. Using Bitmap Index on Ranking Functions for Contextual Ranking Queries

This section presents how to realize the partition-and-prune framework by function constraints and workload-based function constraints, utilizing a novel bitmap index over ranking functions. We also compare with alternative methods based on single-attribute constraints using histogram, B-tree, multi-dimensional index, and bitmap index on single attributes.

5.1. The Method of Using Bitmap Index on Ranking Functions

5.1.1. Intersecting Bitmap Indices on Functions

Following the intuition of using function constraints (Section 4.4), we propose a method of intersecting bitmap indices on functions. Different than conventional bitmap indices, the bitmap indices intersected are built upon ranking functions instead of individual attributes. The motivation in using bitmap index instead of other index structures such as B-tree is that, intersecting bitmap index is much more efficient than intersecting B-trees.

Given a bitmap index [16] on an attribute, there exists a bitmap (a vector of bits) for each distinct attribute value. The length of the vector equals the number of tuples in the indexed relation. In the vector for value x of attribute v, its ith bit is set to 1, when and only when the value of v on the ith tuple is x, otherwise 0. With bitmap indices, complex selection queries can be efficiently answered by bitwise operations (AND, OR, XOR, and NOT) over the bit vectors. Moreover, bitmap indices enable efficient computation of aggregates such as SUM and COUNT [16].

As an efficient index for decision support queries, bitmap index has gained broad interests. Although bitmap index is arguably not as standard as some other index structures such as B-tree, it is widely adopted in DBMSs. Nowadays, bitmap index is supported in major commercial database systems (e.g, Oracle, SQL Server), and it is often the default (or only) index option in column-oriented database systems (e.g., Vertica, C-Store [17], LucidDB), especially for applications with read-mostly or append-only data, such as OLAP and data warehouses.

The state-of-the-art developments of bitmap compression methods [18, 19, 20] and encoding strategies [21, 22, 16] have substantially broaden the applicability of bitmap index on all sorts of attributes. For high-cardinality attributes, a particularly useful type of bitmap index is *bit-sliced index* (BSI) [23]. Given a numeric attribute on integers or floating-point numbers, BSI directly captures the binary representations of attribute values. The tuples' values on an attributes are represented in binary format and kept in s bit vectors (i.e., *slices*), which represent 2^s different values.

For answering contextual ranking queries, 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 query's ranking function. The constraints on buckets are specified by intervals of scores over the chosen functions. Tuples inside a bucket can thus be obtained by intersecting (AND operation) all vectors corresponding to the intervals. A 1 bit in the resulting vector indicates the corresponding tuple is in the bucket. Computing bucket cardinality is a COUNT operation on the resulting vector. The union (OR operation) of all the vectors for candidate buckets produces a single vector, where each 1 bit corresponds to a candidate tuple. Thus we get the IDs of all candidate tuples. The following example illustrates the idea.

Example 7 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 function, i.e., $(w_1, ..., w_n)$, the tuples in the space are ranked in the order of contour lines $w_1 \times a_1 + ... + w_n \times a_n = s$, where each s is a score. We construct a bitmap index for the given function, where each vector corresponds to a score interval.

Figure 5(a) shows a relation R with tuples and attribute values. The ranking function is $x+2y$. Among the functions with bitmap indices constructed, the two functions $x+y$ and $x+3y$ are chosen, since they are close to $x+2y$. Their indices are shown in Figure 5(c) and (d), respectively. Specifically, the index for $x+y$ consists of 4 bitmaps, corresponding to score intervals $[0,2)$, $[2,4)$, $[4,6)$, and $[6,8)$. The index for $x+3y$ also has 4 bitmaps, corresponding to $[0,4)$, $[4,8)$, $[8,12)$, and $[12,16)$. The boundaries between these intervals, i.e., the contour lines, are shown in Figure 5(b). The intersections of these intervals give a tuple space partition. For instance, the bucket corresponding to the intersection of the two shaded areas has constraints $\{2 \leq x+y < 4, 8 \leq x+3y < 12\}$. The upper- and lower-bound scores for this bucket are 6 and 3, respectively, based on linear programming. The bitmaps for $2 \leq x+y < 4$ and $8 \leq x+3y < 12$ are 0001100110 and 0101000100. Therefore the bitmap for the shaded bucket is 0001100110 AND 0101000100 = 0001000100. That is, the bucket contains tuples r_4 and r_8. ■

5.1.2. Heuristics for Choosing Index to Build

We discuss two index selection heuristics. The first heuristic, *random selection*, is to simply 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.

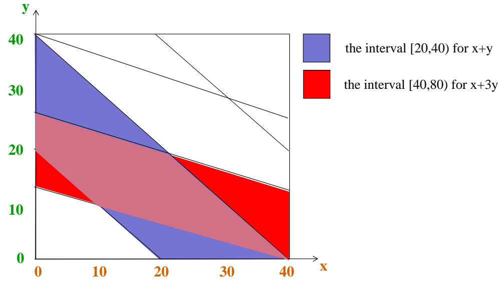
To address this concern, our second heuristic, *workload-based selection*, is to build bitmap indices for those functions that capture the query workload. By doing that, we achieve efficiency for more frequent queries and sacrifice less frequent ones. To be more specific, each linear ranking function is 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. [9] 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 a query space partition is fine-grained enough, queries inside each 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 queries in the space have equal probability to be issued by users, then the strategy degrades to the above random selection heuristic.

[9] The space and buckets of queries should not be confused with the space and buckets of tuples in Section 4.1.

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) Intersecting 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 5: Intersect bitmap indices on functions.

The workload for (contextual) ranking queries is naturally clustered, due to the existence of common senses in user preferences and common interests among similar users.

Questions remain on how many vectors to have for each index. Such a choice is very important in determining the approach's efficiency and it forms a challenging problem that warrants further research. The experiments in Section 6 demonstrate the validity of our approach, under heuristically chosen number of vectors.

5.1.3. Heuristics for Choosing Index to Intersect

With bitmap indices built for the workload, given a new query, we select two indices that are the closest to the query and intersect them. To be more specific, suppose the linear ranking function in a 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 between f and g is defined as their cosine similarity,

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

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

We note that cosine similarity is just an intuitive measure to capture the closeness between functions. Similarly the decision of selecting two indices to intersect is also a heuristic, although in principle we may also use one index or intersect more indices. It warrants further research into various problems, including how to decide the number of intersected indices, how many intervals to build for each index, and how to optimally measure the closeness between functions.

Note that although this method is also based on multi-dimensional space, it does not suffer from the attribute mismatch problem associated with multi-dimensional histogram and index. (More details in Section 5.2.) 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.

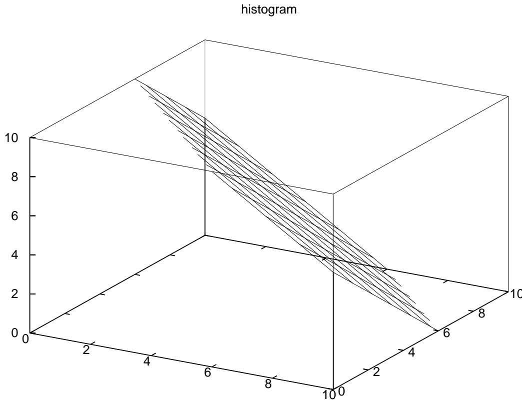


Figure 6: Using MDH to get the rank of a tuple with score $x+y+z=16$.

5.1.4. Dealing with Boolean Conditions

Up to this point, we 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 simplicity of presentation. Before performing the intersections of bit vectors shown in Figure 5, 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. In fact, logic bit vector operations easily allow us to integrate the proposed techniques with bitmap index-based solutions for selection conditions [21, 22, 16] and join queries [24].

5.2. Alternative Methods: Partitioning by Single-Attribute Constraints

In this section we discuss several alternative methods that partition by single-attribute constraints. They may have significant limitations in reality, although they could be competitive in special cases such as very small number of attributes, fixed ranking dimensions, and so on. Nevertheless, they might be the solutions that one resort to at the first glance of the contextual ranking problem. The experimental results in Section 6 verify that these methods have very limited effectiveness.

Using Multi-Dimensional Histogram (MDH): A multi-dimensional histogram (MDH) partitions tuples into buckets, with the cardinality of every bucket pre-computed. Each bucket is defined by intervals (ranges) over individual attributes. For instance, the partition in Figure 1 can indeed be a 2-dimensional histogram. A histogram maintains cardinality information, but cannot provide access to individual tuples. Therefore, we must use multiple SQL range queries concatenated by **UNION**, one for each bucket, to retrieve the tuples in candidate buckets. For instance, suppose Figure 1 is a histogram, the range queries corresponding to the candidate buckets use conditions ($10 \leq a$ **AND** $a<20$ **AND** $30 \leq b$ **AND** $b<40$), ($30 \leq a$ **AND** $a<40$ **AND** $10 \leq b$ **AND** $b<20$), and so on. (An alternative is to use disjunctives to combine these conditions in a single **WHERE** clause.)

This method has serious disadvantages. *First*, the multiple range queries are inefficient to evaluate, as they may require access to the full domain of every attribute. To illustrate, consider Figure 6. 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. *Second*, the dimensions in a histogram may not match the attributes in a ranking function, resulting in loose upper-bound and lower-bound. The loose bounds further produce significant overlapping among the bounds of buckets, thus large number of candidate buckets. For instance, suppose the histogram in Figure 1 is used to answer another query with ranking function $a+b+c$, instead of $a+b$. Attribute c has domain $[0, 40)$. Since the histogram only uses a and b to partition the space, a constraint, $0 \leq c < 40$, is implicitly given for every bucket. With such an identical loose constraint, all buckets may become candidates. *Finally*, it is difficult to apply this method when there are join or selection conditions in our query. Although there exist multi-dimensional histograms for joins, the combination of ranking and Boolean attributes from joined tables will only increase the number of dimensions that need to be matched, resulting in even looser bounds.

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.

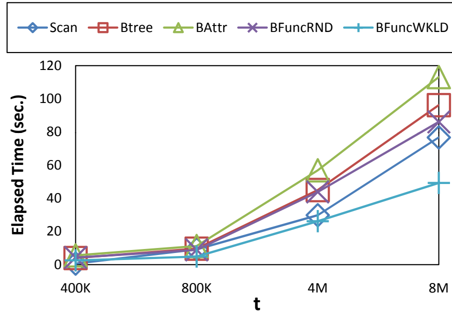
Traversing Multi-Dimensional Index (MDI): Similar to multi-dimensional histogram, MDI is also a partition scheme where buckets are specified by intervals over attributes. Here we use R-tree [25] as an example, without losing generality. The index nodes can be viewed as buckets. There exists a hierarchy in the index tree, thus a hierarchy for the buckets as well. Different than MDH, MDI provides access to tuples. Although standard R-tree does not contain cardinality (number of tuples under each subtree), there are extensions that augment each node in R-tree with cardinality, e.g., aggregate R-tree (aR-tree) [26, 27].

Using MDI has problems similarly existing for MDH. *First,* the attributes in an index may not match the attributes of a ranking function, resulting in loose score bounds, thus a large number of candidate buckets. The mismatch seriously limits the applicability of this method, since it is not affordable to exhaustively build indices for all possible combinations of dimensions. *Second*, it must use random access to tuples to obtain the values of missing ranking attributes (that are not indexed), when un-clustered index is used. The number of random accesses may become prohibitive when the number of candidate buckets increases. *Third*, for MDI with tuple-partitioning (e.g., R-tree), the boundary of an index node (e.g., minimal bounding rectangle (MBR)) is determined towards the efficiency of insertion/deletion operations, which may conflict with the efficiency of answering contextual ranking queries. For instance, an 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. *Finally*, similar to MDH, MDI cannot smoothly deal with selection and join conditions.

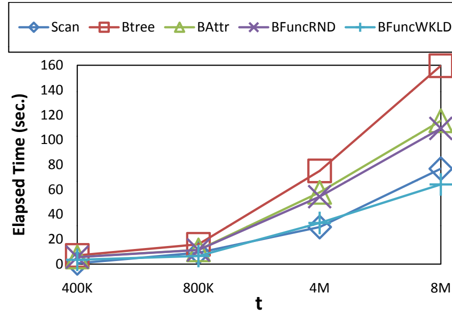
Intersecting B-tree Indices (BTree): We can partition the tuple space by intersecting indices over individual attributes (such as B-trees). [10] A B-tree index naturally partitions an attribute domain into intervals. The results of index intersection provide the (pointers to) tuples in each bucket, thus the cardinality. This method can handle any combinations of ranking attributes as long as the individual indices are available. However, unlike MDI where the space is readily partitioned, it requires explicit intersections of indices. The main overhead of this method thus lies in partitioning, where B-trees over attributes must be fully traversed, and tuple lists are intersected. The traversal on each index may repeat multiple times if the memory cannot hold the nodes from all indices.

Intersecting Bitmap Indices (BAttr): This method intersects bitmap indices instead of B-trees. The procedures of computing cardinalities and retrieving candidates are essentially the same as in the method of function-based bitmap index (Section 5.1). The difference is that bitmap indices here are on individual attributes instead of ranking functions. In other words, it is a special case of function-based bitmap index, since a single attribute can be viewed as a function as well. The buckets are specified by intervals on attributes. The IDs of tuples within one specific interval on an attribute are given by the corresponding bit vector.

[10] Such index intersection was used in evaluating selection queries [28].

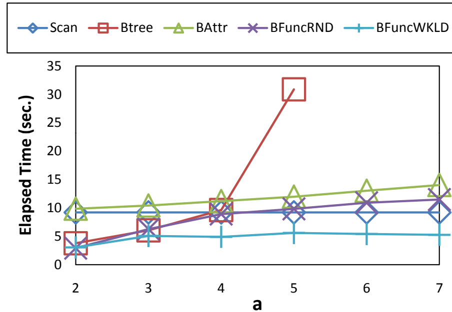


(a) a=4, q=10%.

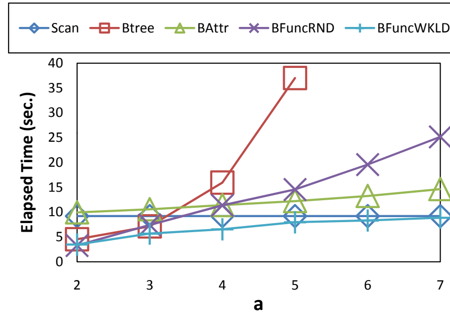


(b) a=4, q=50%.

Figure 7: Single table queries: Execution time varying by t.

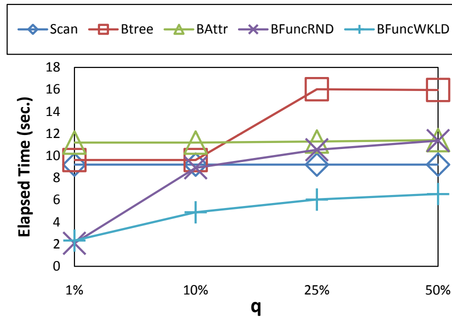


(a) t=800K, q=10%.

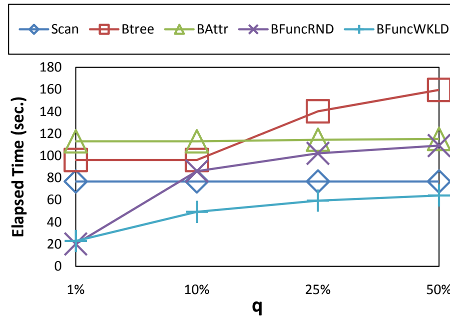


(b) t=800K, q=50%.

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



(a) t=800K, a=4.



(b) t=8M, a=4.

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

The advantage of this method is that the bit operations in getting the vectors for buckets are much more efficient than traversing B-trees and intersecting verbatim lists of tuple IDs. However, in a multi-dimensional space with many intervals on each dimension, the number of buckets and thus the number of bitmap operations can be fairly large, resulting in large cost in computing cardinalities of buckets.

6. Experiments

The algorithms are implemented in C++. Our bitmap index implementation is based on [29], which builds multiple bitmap indices at different domain resolutions and compresses them using the WAH compression method [18]. The B-tree index intersection algorithm is built upon a B-tree implementation in *libgist*, an library that implements GiST [30].

We experimentally compared the following methods: an exhaustive method that uses sequential scan on single table (*Scan*), an 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 the method of intersecting bitmap index on workload-based functions (*BFuncWKLD*).

6.1. Experimental Settings

The experiments were over synthetic tables. The schema of each table consists of a set of attributes, with total width of 100 bytes. The attributes are 4-byte floating point numbers, independently generated by different distributions, including uniform, Gaussian, and cosine distributions.

We also experimented 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 referencing them. A, B, and C have the same size. About half of A's tuples do not join with any tuple in B. Each tuple in the remaining half in average joins with 2 tuples in B. The same applies to the join between C and B.

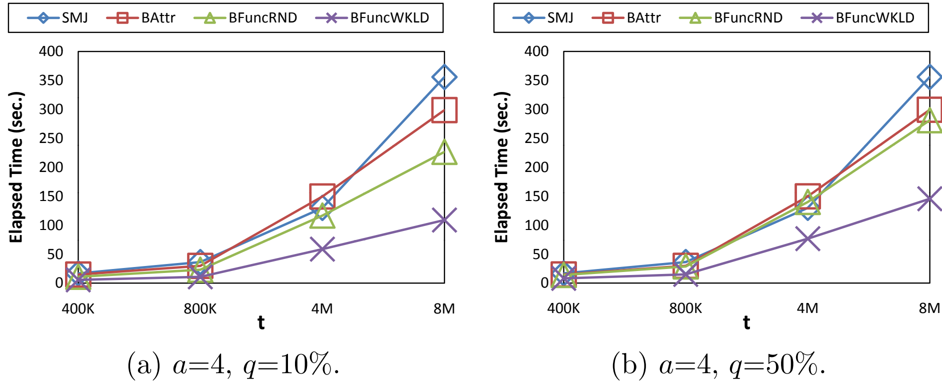
Our queries use weighted-sum over ranking attributes as the ranking function. We experimented with various numbers of ranking attributes. The workload was created by a data generator for clustering algorithms from [31]. Viewing each query ranking function as a point, i.e., a vector of weights, in the query space, a 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 on a weight parameter are generated by a Gaussian distribution with the 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 consisting of 3 146GB SCSI disks, running Linux 2.6.15.

6.2. Experimental Results

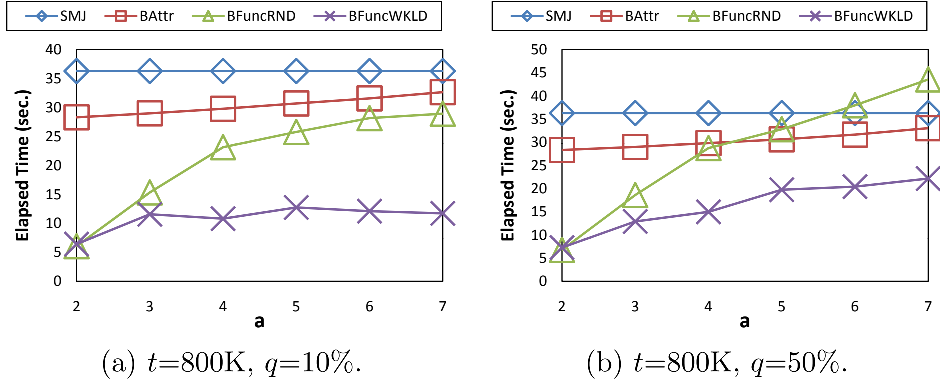
We evaluated the performance of various methods and studied how they are affected by important configuration parameters, which are summarized in Table 1. For *BFuncRND* and *BFuncWKLD*, by default there are bitmap indices built for 200 functions. For *BFuncWKLD*, the functions are chosen based on a query workload containing 500 queries. For each index on a function, we use bit-sliced index (BSI) [23] mentioned in Section 5.1.1. 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 vectors, which represent 2^v ranges. By default v=7.

Single-Table Queries:
To evaluate the performance of single-table queries, we conducted experiments under groups of configurations by the value combinations of three parameters, t, a, and q. In each group of experiments, we varied the value of one parameter and fixed the values of the remaining two. We then ran all the methods and studied how their performance is affected by the varying parameter value. The resulting wall-clock execution time under



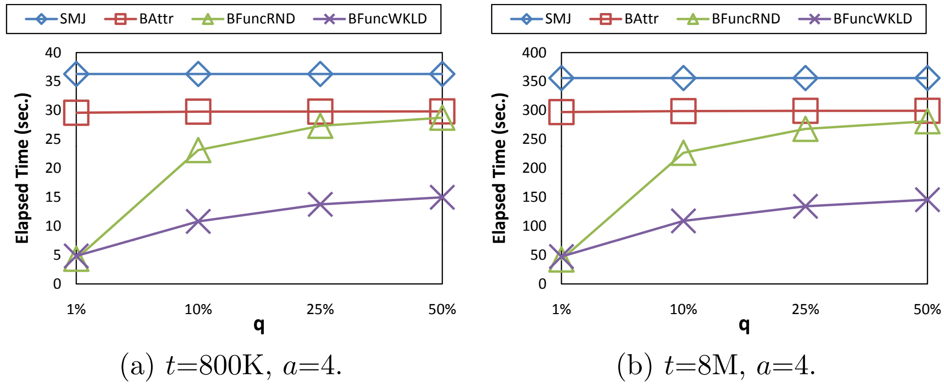
(a) a=4, q=10%. (b) a=4, q=50%.

Figure 10: Join queries: Execution time varying by t.



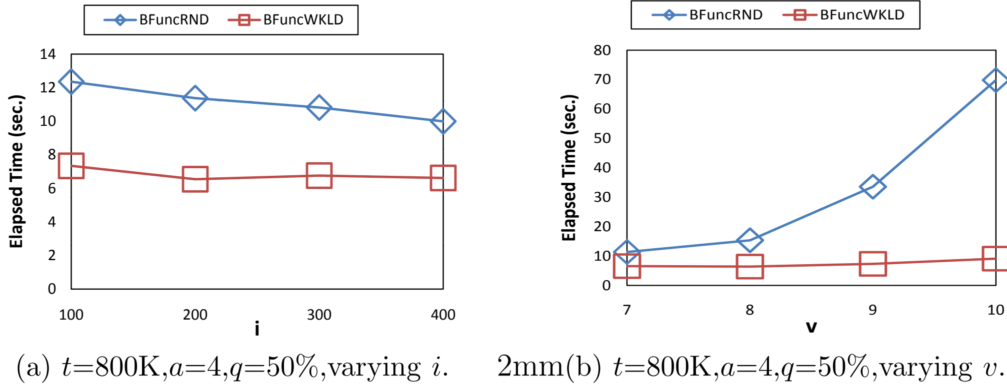
(a) t=800K, q=10%. (b) t=800K, q=50%.

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



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

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



(a) t=800K,a=4,q=50%,varying i. 2mm(b) t=800K,a=4,q=50%,varying v.

Figure 13: Single table queries: varying by i and v.

six sample groups of experiments is shown in Figure 7, 8, and 9. 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 other methods, except *BFuncWKLD*. This verifies the analysis of various methods in Section 5.2. They all have significant disadvantages. For instance, Figure 8 shows that, as the number of ranking attributes increases, the performance of *Btree* degrades exponentially due to the fact that it has to fully traverse all B-trees and intersect their tuple pointers. As another example, Figure 8(b) clearly illustrates the "curse of dimensionality" on *BFuncRND*, as mentioned in Section 5.1.2. The figure also shows that *BFuncWKLD* scales properly by number of ranking attributes. We also note that user-defined ranking functions typically are defined over small number of ranking attributes.

Third, Figure 9 shows that requested rank position has an important impact on algorithm efficiency. Specifically, the smaller rank position, the more efficient *BFuncRND* and *BFuncWKLD* are. On the other hand, the performance of the straightforward approach *Scan* is independent from rank position. Figure 9 shows that, to obtain the rank of an object ranked at 1% (e.g., the object ranked at 8192 when t=800K), *BFuncRND* outperformed *Scan*. However, as rank position increases, it became worse than *Scan*. With regard to *BFuncWKLD*, it always outperformed *Scan*, but its margin of advantage over *Scan* shrunk as rank position increases.

Join Queries:
The results of join queries are shown in Figure 10-12. Note that *SMJ* replaces *Scan* for join queries and *Btree* becomes unapplicable. We make the following observations from these figures: *First*, *BFuncWKLD* is still clearly the best method, and its advantages over other algorithms are enlarged under join. *Second*, different from single-table scenario, the exhaustive approach *SMJ* now is often the worst method. This is due to that a full join needs to process a large number of input tuples and intermediate results. *Third*, *BFuncRND* is often the second best method. However, when the number of ranking attributes increases, this method suffers more than other methods, as shown by Figure 11.

To further understand the methods *BFuncRND* and *BFuncWKLD*, we conducted more experiments to analyze how their performance is affected by various parameters, as shown in Figure 13. As expected, when the number of built indices increases (Figure 13(a)), these two methods are more efficient. However, 400 indices do not give us substantial performance improvement over 100 indices. This indicates that a small number of indices are sufficient for the given workload. Under other workloads, more indices may become necessary. Figure 13(b) shows that increasing the number of vectors makes the performance of *BFuncRND* worse. The reason is that randomly se-

lected 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 and 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 a small number of intersections and candidate buckets.

Note that we did not experiment with Boolean selection and join conditions. The synthetic table can be viewed as the results after such conditions are applied. To obtain the results, the approach of using bitmap index has been well-studied and is shown to be very efficient for range selections and star-joins [24, 16, 21, 22]. In Section 5.1.4 we have discussed how to integrate with such techniques. Therefore, to focus on the performance study of contextual ranking queries, we do not mix with the performance measurements on Boolean conditions, whose results are well-known.

7. Conclusion

We studied contextual ranking queries that obtain the ranks of query objects among context objects. Such queries are useful in many data exploration applications. For processing contextual ranking queries, we develop a general partition-and-prune framework and a novel method based on bitmap index on ranking functions. We analytically study the cost model of this framework and empirically compare various methods. Experiments show that our method can be substantially more efficient than alternative methods.

Acknowledgement

We thank Kevin Chang and Ihab Ilyas for discussion on this work.

References

[1] R. Fagin, A. Lote, M. Naor, Optimal aggregation algorithms for middleware, in: PODS, 2001.

[2] I. F. Ilyas, G. Beskales, M. A. Soliman, A survey of top-k query processing techniques in relational database systems, ACM Comput. Surv. 40 (2008) 11:1–11:58.

[3] W. Kießling, Foundations of preferences in database systems, in: VLDB, 2002, pp. 311–322.

[4] S. Börzsönyi, D. Kossmann, K. Stocker, The skyline operator, in: Proceedings of the 17th International Conference on Data Engineering, 2001, pp. 421–430.

[5] J. Chomicki, Preference formulas in relational queries, ACM Trans. Database Syst. 28 (2003) 427–466.

[6] R. Agrawal, A. Swami, A one-pass space-efficient algorithm for finding quantiles, in: Proc. 7th Int. Conf. Management of Data, COMAD, 1995.

[7] A. Arasu, G. S. Manku, Approximate counts and quantiles over sliding windows, in: PODS, 2004.

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

[9] M. L. Yiu, N. Mamoulis, Y. Tao, Efficient quantile retrieval on multi-dimensional data., in: EDBT, 2006.

[10] C. Li, Enabling data retrieval: By ranking and beyond, Ph.D. thesis, University of Illinois at Urbana-Champaign (2007).

[11] X. Lian, L. Chen, Probabilistic inverse ranking queries in uncertain databases, The VLDB Journal 20 (2011) 107–127.

[12] T. Bernecker, H.-P. Kriegel, N. Mamoulis, M. Renz, A. Zuefle, Continuous inverse ranking queries in uncertain streams, in: SSDBM, 2011, pp. 37–54.

[13] Q. Wan, R. C.-W. Wong, I. F. Ilyas, M. T. Özsu, Y. Peng, Creating competitive products, VLDB (2009) 898–909.

[14] D. Bertsimas, Nonlinear Programming, 2nd Edition, Athena Scientific, Belmont, Massachusetts, 1995.

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

[16] P. E. O'Neil, D. Quass, Improved query performance with variant indexes, in: SIGMOD, 1997, pp. 38–49.

[17] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, S. Zdonik, C-store: A column oriented dbms, in: VLDB, 2005.

[18] K. Wu, E. J. Otoo, A. Shoshani, Optimizing bitmap indices with efficient compression, ACM TODS 31 (1) (2006) 1–38.

[19] G. Antoshenkov, Byte-aligned bitmap compression, in: Proceedings of the Conference on Data Compression, 1995.

[20] T. Johnson, Performance measurements of compressed bitmap indices, in: VLDB, 1999, pp. 278–289.

[21] C. Y. Chan, Y. E. Ioannidis, An efficient bitmap en-

coding scheme for selection queries, in: SIGMOD, 1999.

[22] M.-C. Wu, A. P. Buchmann, Encoded bitmap indexing for data warehouses, in: ICDE, 1998, pp. 220–230.

[23] D. Rinfret, P. O'Neil, E. O'Neil, Bit-sliced index arithmetic, in: SIGMOD, 2001, pp. 47–57.

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

[25] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: SIGMOD, 1984, pp. 47–57.

[26] I. Lazaridis, S. Mehrotra, Progressive approximate aggregate queries with a multi-resolution tree structure, in: SIGMOD, 2001.

[27] D. Papadias, P. Kalnis, J. Zhang, Y. Tao, Efficient OLAP operations in spatial data warehouses, in: SSTD, 2001.

[28] C. Mohan, D. J. Haderle, Y. Wang, J. M. Cheng, Single table access using multiple indexes: Optimization, execution, and concurrency control techniques, in: EDBT, 1990.

[29] R. R. Sinha, S. Mitra, M. Winslett, Bitmap indexes for large scientific data sets: A case study, in: IPDPS, 2006.

[30] J. M. Hellerstein, J. F. Naughton, A. Pfeffer, Generalized search trees for database systems, in: VLDB, 1995.

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