INTEGRATING OLAP AND RANKING: THE RANKING-CUBE METHODOLOGY

BY

DONG XIN

B.Eng., Zhejiang University, 1999
M.S., Zhejiang University, 2002

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Abstract

Recent years have witnessed an enormous growth of data in business, industry, and Web applications. Database search often returns a large collection of results, which poses challenges to both efficient query processing and effective digest of the query results. To address this problem, ranked search has been introduced to database systems. We study the problem of On-Line Analytical Processing (OLAP) of ranked queries, where ranked queries are conducted in the arbitrary subset of data defined by multi-dimensional selections. While pre-computation and multi-dimensional aggregation is the standard solution for OLAP, materializing dynamic ranking results is unrealistic because the ranking criteria are not known until the query time. To overcome such difficulty, we develop a new ranking cube method that performs semi off-line materialization and semi online computation in this thesis. Its complete life cycle, including cube construction, incremental maintenance, and query processing, is also discussed. We further extend the ranking cube in three dimensions. First, how to answer queries in high-dimensional data. Second, how to answer queries which involves joins over multiple relations. Third, how to answer general preference queries (besides ranked queries, such as skyline queries). Our performance studies show that ranking-cube is orders of magnitude faster than previous approaches.

To my daughter Karen, my wife Xu and my parents.

Acknowledgments

This thesis would not have been possible without the generous help and consistent support of my thesis advisor, Professor Jiawei Han. I would like to express my appreciation for his valable insights, encouragement, and thoughtful discussions. Professor Han not only helped me on the research project in the field of data mining and database management, but also taught me to be confident, persistent in performing research. I sincerely believe I have benefited tremendously from his help and impact on me during my years under his supervision.

I would like to thank Professor Bruce R. Schatz, Professor Kevin C.-C. Chang and Dr. Venkatesh Ganti, for taking their valuable time to be on my thesis committee and for providing many helpful comments and suggestions.

I would also like to thank Deng Cai, Chen Chen, Hong Cheng, Jing Gao, Hector Gonzalez, Chul Yun Kim , Sangkyum Kim, Xiaolei Li, Chao Liu, Zheng Shao, Tianyi Wu, Xifeng Yan, Xiaoxin Yin, Feida Zhu and all other group members in our research group for providing insightful comments on my work and for providing a friendly environment to work in.

I am indebted to my wife and my parents for their continuous love, support and understanding.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation of this research

Recent years have witnessed an enormous growth of data in business, industry, society, Web, and scientific applications. There is an imminent need for effective and scalable methods for analyzing and exploring this overwhelming large and complex data. The goal of this research is to develop query processing methods to facilitate effective and efficient information exploration. More specifically, for a given query, instead of deriving the complete set of answers, it is often desirable to derive only a subset but high-quality data records based on user's preference. A commonly used strategy is to rank the data records and only return top-k answers. A typical application is database-centric web search, such as product search on bizrate.com, apartment search on apartments.com, and used car search on kbb.com.

A top-k query only returns the best k results according to a user-specified preference, which generally consists of two components: a *multi-dimensional selection condition* and a *dynamic ranking function*. With the mounting of an enormous amount of data in business, ranking with respect to multi-dimensional group-bys becomes prominent for effective data analysis and exploration. Example application scenarios are illustrated as follows.

Example 1. (Multi-dimensional data exploration) Consider an online used car database R that maintains the following information for each car: type (*e.g.*, sedan, convertible), maker (*e.g.*, Ford, Hyundai), color (*e.g.*, red, silver), transmission (*e.g.*, auto, manual), price, milage,

etc.. Two typical top-k queries over this database are:

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

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

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

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

As shown in the above examples, different users may not only propose *ad hoc* ranking functions but also use different interesting subset of data. In fact, in many cases, users may want to have a thorough study of the data by taking a *multi-dimensional analysis* of the

top-k query results.

Example 2. (Multi-dimensional data analysis) Consider a notebook comparison database (*e.g.*, bizrate.com) with schema (*brand, price, CPU, memory, disk*). Suppose a function f is formulated on CPU, memory and disk to evaluate the market potential of each notebook. An analyst who is interested in dell low-end notebooks may first issue a top-k query with *brand = "dell"* and *price* ≤ 1000, and then rolls up on the *brand* dimension and checks the top-k low-end notebooks by all makers. By comparing two sets of answers, the analyst will find out the position of dell notebooks in the low-end market. ■

The above application scenarios propose a new challenging task for database system: *How to efficiently process top-k queries with multi-dimensional selection conditions and ad hoc ranking functions.* Given the sample queries in the above examples, current database systems will have to evaluate all the data records and output those top-k results which satisfy the selection conditions. Even if indices are built on each selection dimension, the database executer still needs to issue multiple random accesses on the data. This is quite expensive, especially when the database is large.

Efficient processing such queries requires the database system to simultaneously push both boolean predicates and preference analysis deep into search. There has been fruitful research work on efficiently processing top-k queries [7, 20, 17, 28, 30, 16, 41, 45] in database systems. However, all these studies are performed under the context that the query consists of ranking functions only. By introducing boolean predicates in the query, many algorithms need to be carefully re-examined.

On the other hand, current database management systems execute ranked queries with boolean predicates by first retrieve data objects according to boolean selection conditions, and then conduct the ranking analysis. This approach is not efficient, especially when the database is large and the number of output is small. In conclusion, existing solutions conduct either boolean-only or ranking-only search, and thus are not satisfactory.

For multi-dimensional analysis, data cube [27] has been extensively studied. Material-

ization of a data cube is a way to pre-compute and store multi-dimensional aggregates so that online analytical processing can be performed efficiently. Traditional data cubes store the basic measures such as SUM, COUNT, AVERAGE, *etc.*, which are not able to answer complicated top-k queries.

To meet the requirement of online analytical processing, the database system has to return the user-preferred answers from any data groups, in a very efficient way. The dynamic nature of the problem imposes a great challenge for the database research community. To the best of our knowledge, the problem of efficient processing top-k queries with multi-dimensional selection conditions is not well addressed yet. In this thesis, we address this problem from an integrated viewpoint. On the one hand, online analytical query processing requires off-line pre-computation so that multi-dimensional analysis can be performed on the fly; on the other hand, the ad-hoc ranking functions prohibit full materialization. A natural proposal is to adopt a *semi off-line materialization* and *semi online computation* model. This thesis discusses the design principles, implementation issues and various extensions.

1.2 Problems addressed

We start from the basic problem on searching ranked results in a single relation with limited number of dimensions. We then extend the problem in three dimensions: query on high-dimensional data; query over multiple relations; and query with other types of preference criteria.

1.2.1 Ranked Query with Selections

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

function. A possible SQL-like notation for expressing top-k queries is as follows:

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

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

We further notate A_i ($i = 1, \ldots, S$) as *selection dimension* (or *boolean dimension*) and N_i ($1, \ldots, R$) as *ranking dimension*. In general, a real valued attribute can also be a selection dimension if it is discretized. A categorical attribute can also be a ranking dimension if the distance is defined on the values in the domain. In this thesis, we first assume the number of ranking dimensions is relatively small (*e.g.*, 2-4), while the number of selection dimensions could be rather large (*e.g.*, more than 10). This is a typical setting for many real applications. For example, both the used car database and the hotel database only have a limited number of ranking dimensions (*e.g.*, price, milage, distance). While the number of selection dimensions is much larger. We extend our solution to cases where the number of ranking dimensions is also large in Chapter 5.

For simplicity, we first demonstrate our method using the *convex* ranking functions. The formal definition of the convex function is presented in Definition 1.

Definition 1 *(**Convex Function** [55]) A continuous function f is convex if for any two points x_1 and x_2 in its domain $[a, b]$, and any λ where $0 < \lambda < 1$:*

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

The convex functions already cover a broad class of commonly used functions. For exam-

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

We will relax the constraint to a more general type of functions: **Lower-bound Function**. Without losing generality, we assume that users prefer *minimal* values. The query results are a set of objects that belong to the data set satisfying the boolean predicates, and are also ranked high (for top-k) in the same set. We assume that the ranking function f has the following property: *Given a function $f(N'_1, N'_2, \ldots, N'_j)$ and the domain region Ω on its variables, the lower bound of f over Ω can be derived.* For many continuous functions, this can be achieved by computing the derivatives of f.

1.2.2 Extensions

High-dimensional Data

In some applications (such as apartment search), the number of dimensions is large. We will address two scenarios:

- The number of selection dimensions is large. For instance, in the apartment search application, the search criteria could be whether the apartment has In-Unit Washer/Dryer, or Washer/Dryer Hookups, or Laundry Room; whether the apartment has air-condition, walk-in closets, hardwood Floors; whether the community has parking available, fitness center, pool...

- The number of ranking dimensions is large. Use the apartment search as example, the ranking criteria includes the rent rate, the square footage, the distance to some interesting place (such as shopping mall, park, beach, *etc.*), the number of bedrooms,

the number of bathrooms, the difference between the available date and move in date, the application fee, the deposit fee...

High boolean dimension suffers *the curse of dimensionality* in cube materialization, and high ranking dimension leads to the difficulty to effectively partition and search data. To overcome this difficulty, advanced solutions are needed.

Ranked Query over Multiple Relations

When multiple relations exist in the database, ranked queries can be issued crossing relations. A multi-relational ranked query consists of a join condition, a set of boolean dimensions and ranking dimensions from each participating relation. The result of a top-k query is an ordered set of k tuples that is filtered by the boolean constraints and ordered according to the given ranking function.

General Preference Queries

Top-k queries are related to several other preference queries, such as skyline query [12] and convex hulls [11]. Skyline query asks for the objects that are not dominated by any other object in all dimensions. A convex hull query searches a set of points that form a convex hull of all the other data objects. We will also discuss how to apply our proposed methods to efficiently answer general preference queries with multi-dimensional selections.

1.3 Proposed approaches

1.3.1 Integrating Ranking and Selection

We propose to *simultaneously push boolean and ranking pruning in query processing.* More specifically, to address multi-dimensional boolean selections, we adopt the data cube model [27], which has been popularly used for fast and efficient on-line analytical processing (OLAP)

in multi-dimensional space. To support ad hoc ranking analysis, we partition the data according to measure attributes. To seamlessly integrate multi-dimensional selection and ranking analysis into a single framework, we develop a new materialization model, called *ranking-cube*, which summarizes data partition for each multi-dimensional selection condition (*e.g.*, a cell in the data cube). Ranking-cube does not provide query answers directly. Instead, it is used to aid efficient query processing. Our experimental results show that the new method is at least one order of magnitude faster than the previous approaches.

According to different approaches in searching for top ranked results, we demonstrate this ranking-cube framework by two possible implementations: the *grid partition* with *neighborhood search*, and the *hierarchical partition* with *top-down search*.

It is also desirable to have an incremental methodology to efficiently maintain the ranking cube with the data insertion or deletion. We develop slightly different incremental update methods based on above two implementations. For grid partition, one can temporally allocate new data according to pre-computed blocks, and re-partition the data periodically. For hierarchical partition, incremental data update on data partition can be first applied, and the structure changes can be propagated to ranking cube. In this thesis, we mainly elaborate our incremental solution with hierarchical solution.

1.3.2 Towards High-dimensional Data

In some applications, the number of dimensions is large. We discuss the possible solutions for the high boolean dimensions and high ranking dimensions. With high boolean dimensions, a full materialization of the ranking cube is too space expensive. Observing that many real life ranked queries are likely to involve only a small subset of attributes, we can carefully select the cuboids which needs to be materialized, where the space requirement for materialization grows linearly with the number dimensions.

With high ranking dimensions, any single partition is not effective. A possible solution is to create multiple data partitions, each of which consists a subset of ranking dimensions. The

query processing thus conducts search over a joint space involving multiple data partitions. Our solution further addresses two challenges: (1) the search space for f grows exponentially with the number of partitions, and (2) the satisfactory of boolean predicates for a joint block is difficult to determine.

1.3.3 Extending to Multi-Relations

When multiple relations exist in the database, a slight variation of ranking cube can be built for each relation. A multi-relational ranked query that consists of a join condition and a set of boolean dimensions and ranking dimensions from each participating relation. We extend the original ranking cube framework to multiple relations (*i.e.*, multiple ranking cubes). The partition is performed on each relation, and the ranking cube is built with respect to the boolean dimensions in each relation. Furthermore, we propose a complete system to support ranked queries over multiple relations.

1.3.4 Adapting to other Preference Queries

Top-k queries are related to several other preference queries, such as skyline query [12] and convex hulls [11]. Skyline query asks for the objects that are not dominated by any other object in all dimensions. A convex hull query searches a set of points that form a convex hull of all the other data objects. The ranking cube framework is also applicable to these queries. For instance, the convex hull query algorithm developed by [11] progressively retrieves R-tree blocks until the final answers are found. Ranking cube can be integrated into this search procedure by pruning some blocks that do not contain tuples satisfying boolean predicates.

1.4 Organization of the Thesis

The remainder of this thesis are organized as follows.

- Chapter 2 discusses the related work. Our work is mainly related to (1) ranked query processing; and (2) OLAP and data cube.

- Chapter 3 presents the main idea of ranking-cube. The framework is demonstrated by a simple grid partition (in materialization) and neighborhood search (in query processing).

- Chapter 4 advances the ranking-cube model by an alternative hierarchical partition method (in materialization) and top-down search (in query processing). We further demonstrate the off-line cube computation and incremental update methods in this framework.

- Chapter 5 addresses the difficulty with high ranking dimensions (The problem with high boolean dimensions is discussed in Chapter 3). Our solution uses an index (partition) merge framework.

- Chapter 6 further extends our solutions in Chapter 3 and Chapter 5 to ranked query processing over multiple relations. Specially, we present a solution for *SPJR* (*i.e.*, selection, projection, join and ranked) queries.

- Chapter 7 applies the ranking cube with general preference queries. Typically, we demonstrate the solution on skyline and dynamic skyline queries.

- Chapter 8 concludes our study.

Chapter 2

Related Work

Our work is tightly related to OLAP (On-Line Analytical Processing) and ranked query processing. OLAP and ranking are currently separate technologies in the database systems. OLAP refers to a set of data analysis tools developed for analyzing data in data warehouses since 1990s [27]. A data warehouse stores a multi-dimensional, logical view of the data, and supports management's decision-making process. A point in a data cube stores a consolidated measure of the corresponding dimension values in a multi-dimensional space. OLAP operations, such as drill-down, roll-up, pivot, slice, and dice, are the ways to interact with the data cube for multi-dimensional data analysis. Ranking is a way to filter the query results and retain partial but high-quality answers. With the increasing integration of the database systems with Web search, information retrieval, multimedia, and data mining applications, database query processing has been evolving from finding the complete set of answers to finding top-k answers. This leads to the popularity in research into ranked query processing [7, 22, 21, 20, 17, 28, 30, 40, 45, 41, 64]. In the rest of this chapter, we discuss the previous work on ranked query processing and data cube.

2.1 Ranked Query Processing

Our framework emphasizes on how to design off-line materialization to speed up the online query processing. Previous work on rank-aware materialization mainly focuses on linear ranking functions; and previous work on rank-aware query processing are mainly derived from the threshold algorithm [30].

2.1.1 Rank-aware Materialization

Recent successful work on rank-aware materialization includes layered and non-layered indices (views) for linear ranking functions.

The non-layered approaches includes the PREFER system [40], where tuples are sorted by a pre-computed linear weighting configuration. Queries with different weights will be first mapped to the pre-computed order and then answered by determining the lower bound value on that order. When the query weights are close to the pre-computed weights, the query can be answered efficiently. Otherwise, this method may retrieve more data tuples. An example is shown below.

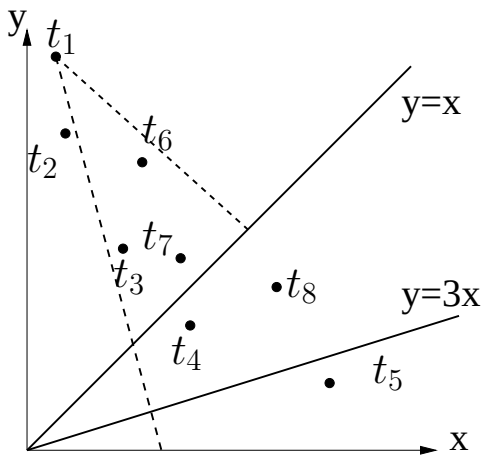


Figure 2.1: **Rank mapping in PREFER**

Example 1 *Fig. 2.1 shows 8 tuples:* t_1, $t_2, \ldots, t_8$ *in a tiny database. Each tuple has two attribute* x *and* y*. Suppose the pre-computed ranking order is built by the ranking function* $x + y$*. The order of each tuple in the index is determined by its projection onto the line* $y = x$*, which is orthogonal to* $x + y = 0$*. Similarly, a query with ranking function* $3x + y$ *corresponds to the line* $y = 3x$*. Suppose the query asks for top-1, and the results are* t_2*. The system will retrieve all tuples in the database ranked before* t_2 *with respect to* $x + y$*.*

The layered indexing methods includes the Onion technique [21] and the Robust Indexing [68] Generally, they organize data tuples into consecutive layers according to the geometry

layout, such that any top-k queries can be answered by up to k layers of tuples. Thus the worst case performance of any top-k query can be bounded by the number of tuples in the top k layers. The Onion technique [21] greedily computes convex hulls on the data points, from outside to inside. Each tuple belongs to one layer. The query processing algorithm is able to leverage the domination relationships between consecutive layers and may stop earlier than the k^{th} layer is touched. For example, if the best rank of tuples in the c^{th} layer is no less than k among all tuples in the top-c layers ($c \leq k$), then all the tuples behind the c^{th} layer need not to be touched because they cannot rank before k. However, in order to leverage this domination relation between the consecutive layers, each layer is constructed conservatively and some tuples are unnecessarily to be put in top layers (as demonstrated in example 2).

The Robust Indexing exploits the fact that it may be beneficial to study more tuple-wise *domination relations*. A set of tuples S *dominates* a tuple t if for any query, there is at least one tuple in S ranks higher than t. A tuple t can be at least put in $(n+1)^{th}$ layer if there are n exclusive sets which dominate t. A good indexing strategy should push a tuple as *deeply* as possible so that it has less chance to be touched in query execution. The Robust Indexing uses an alternative criterion for sequentially layered indexing: for any k, the number of tuples in top k layers is minimal in comparison with all the other layered alternatives. Since any top-k query can be answered by at most k layers, this proposal aims at minimizing the worst case performance on any top-k queries. Hence the proposed index is *robust*.

Example 2 *Suppose all query weights are non-negative. We can improve the onion technique by constructing convex shells, instead of convex hulls for each layer. Using the same sample database in Example 1, the constructed convex shells are shown in Fig. 2.2 (a). There are two layers constructed on the 8 tuples and for any top-2 query, all 8 tuples will be retrieved. Robust index exploits more layer opportunities in this example. Fig. 2.2 (b) shows the robust index construction which has four layers:* $\{t_1, t_2, t_3, t_4, t_5\}$, $\{t_7\}$,$\{t_8\}$ *and*

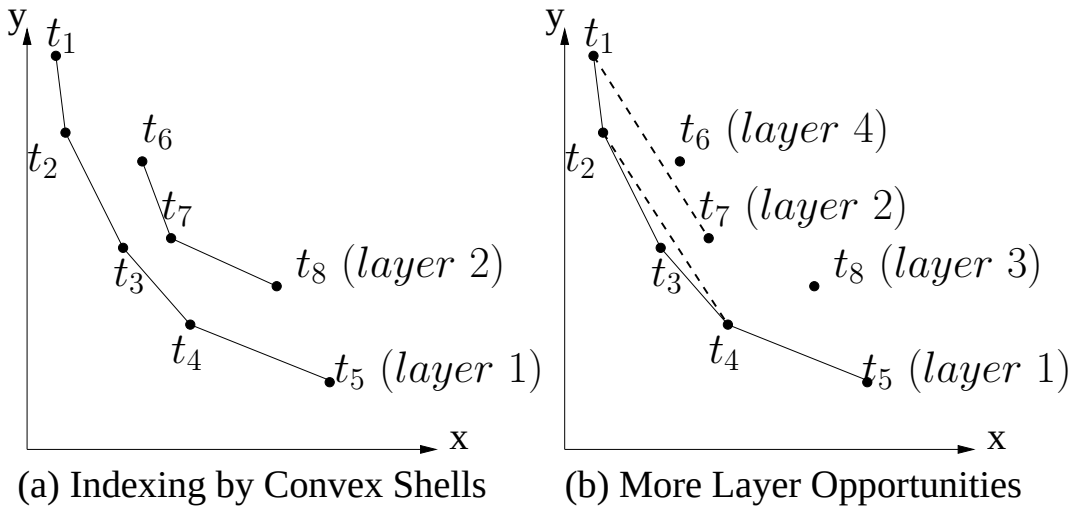


Figure 2.2: **Multi-Layered Index**

$\{t_6\}$ (solid lines). Tuple t_6 can be put in the 4^{th} layer because for any linear query with non-negative weights, t_3 must rank before t_6, one of the tuples in $\{t_2, t_4\}$ must rank before t_6 and one of the tuples in $\{t_1, t_7\}$ must rank before t_6 (dashed lines). These claims can be verified by linear algebra. For the same reason, t_8 can be put in 3^{rd} layer. Any top-2 query on this layered index will only retrieve 6 tuples.

2.1.2 Rank-aware Query Processing

Most rank-aware query processing follows the family of threshold algorithms (*i.e.*, TA) [28, 30, 29]. TA uses a *sort-merge* framework that sequentially scans a set of pre-computed lists, each of which is sorted according to individual scores, and merges data objects from different lists to compute the aggregated scores. Usually, it stops scanning long before reaching the end of the lists. TA has been extensively studied in the context of middle-ware, information retrieval, and multimedia similarity search, where the aggregation functions are usually *monotone*. We say that a function f is monotone if $f(x_1, x_2, \ldots, x_m) \leq f(x'_1, x'_2, \ldots, x'_m)$ whenever $x_i \leq x'_i$, for every i.

The original scheduling strategy for TA-style algorithms is round-robin over all lists (mostly to ensure certain theoretical properties). Early variants also made intensive use of

random access (RA) to index entries to resolve missing score values of result candidates, but for very large index lists with millions of entries that span multiple disk tracks, the resulting random access cost is much higher than the cost of a sorted access (SA). To remedy this, [29] introduced a combined algorithm (CA) framework

In the TA framework, there are several other variations. Chang et al. [20] developed the MPro method. This method relies on some standard Threshold algorithm to compute a list of candidates and their scores for the indexed attributes of a query. Based on that list, the top-k answer to the query is then computed by scheduling additional RAs on the non-indexed attributes, prioritizing top-k candidates based on their best scores (i.e., upper bounds of their true total scores). An extended method estimates, based on single-attribute selectivities, the probability that a single predicate achieves a score that would qualify the data item for the top-k result.

Bruno et al. [15] developed the Upper method that conceptually alternates between RA and SA steps. For RA scheduling, Upper selects the data item with the highest best score and performs a single RA on the attribute (source) with the highest expected score. This is repeated until no data item remains that has a higher best score than any yet unseen document could have; then SAs are scheduled in a round-robin way until such a data item appears again.

Bruno et al. [15] also developed the Pick method that runs in two phases: in the first phase, it makes only SAs until all potential result documents have been read (i.e., as soon as the best score that a yet unseen document could have is not larger as the current kth largest partial score of an already seen document). In the second phase, it makes RAs for the missing dimensions of candidates that are chosen similarly to Upper, taking the (source specific) costs of RAs and the expected score gain into account.

The rank-aware materialization focuses on linear ranking functions, and the rank-aware query processing is confined to monotone functions. Hence, they have limitations to answer other common ranking functions. Moreover, those approaches are not aware of the multi-

dimensional selection conditions.

2.2 OLAP and Data Cube

On Line Analytical Processing, or OLAP, is an approach to quickly providing answers to analytical queries that are multidimensional in nature. Databases configured for OLAP employ a multidimensional data model, allowing for complex analytical and ad-hoc queries with a rapid execution time.

2.2.1 Concept and Definitions

In the core of any OLAP system is a concept of an data cube. It consists of numeric facts called measures which are categorized by dimensions. The cube meta data is typically created from a star schema or snowflake schema of tables in a relational database. Measures are derived from the records in the fact table and dimensions are derived from the dimension tables.

Since the introduction of data warehousing, data cube, and OLAP [33], efficient computation of data cubes has been one of the focusing points in research with numerous studies reported. The research work can be classified into the following categories: (1) efficient computation of full or iceberg cubes with simple or complex measures [1, 76, 54, 9, 37], (2) selective materialization of views [38, 4, 34, 35, 59], (3) computation of compressed data cubes by approximation, such as quasi-cubes, wavelet cubes, etc. [5, 66, 57, 6], (4) computation of condensed, dwarf, or quotient cubes [43, 67, 61, 44], and (5) computation of stream "cubes" for multi-dimensional regression analysis [24].

Among these categories, the first one, efficient computation of full or iceberg cubes, plays a key role because it is a fundamental problem. The problem of cube computation can be defined as follows.

Definition 2 *(Group-By Cell) In an n-dimension data cube, a cell* $c = (a_1, a_2, \ldots, a_n : m)$

(where m is a measure) is called a k-dimensional group-by cell *(i.e., a cell in a k-dimensional cuboid), if and only if there are exactly k ($k \leq n$) values among $\{a_1, a_2, \ldots, a_n\}$ which are* not $*$. *We further denote $M(c) = m$ and $V(c) = (a_1, a_2, \ldots, a_n)$.*

Definition 3 *(Iceberg Cell) Given a threshold constraint on the measure, a cell is called* iceberg cell *if it satisfies the constraint. A popular iceberg constraint on measure count is $M(c) \geq$* min_sup*, where* min_sup *is a user-given threshold.*

Definition 4 *(Closed Cell) Given two cells $c = (a_1, a_2, \ldots, a_n, m)$ and $c' = (a'_1, a'_2, \ldots, a'_n, m')$, we denote $V(c) \leq V(c')$ if for each a_i $(i = 1, \ldots, n)$ which is not $*$, $a'_i = a_i$ (the equality holds iff $V(c) = V(c')$). A cell c is said to be* covered *by another cell c' if $\forall c''$ such that $V(c) \leq V(c'') < V(c')$, $M(c'') = M(c')$. A cell is called a* closed cell *if it is not covered by any other cells.*

2.2.2 Data Cube Computation

Here we review some representative algorithms for iceberg cube and closed cube computation. Previous studies for iceberg cube computation have developed three major approaches, top-down, bottom-up, and integrated. The top-down approach is represented by the Multi-Way Array Cube (called MultiWay) algorithm [76], aggregates simultaneously on multiple dimensions; however, it cannot take advantage of Apriori pruning when computing iceberg cubes. The bottom-up approach is represented by BUC [9], that computes the iceberg cube bottom-up and facilitates Apriori pruning. The integrated approach is represented by Star-Cubing that combines both top-down and bottom-up computational order, and leverages both computational benefits. For closed cube computation, we review the C-Cubing algorithm, which is built based on Star-Cubing.

MultiWay

MultiWay [76] is an array-based top-down cubing algorithm. It uses a compressed sparse array structure to load the base cuboid and compute the cube. In order to save memory usage, the array structure is partitioned into chunks. It is unnecessary to keep all the chunks in memory since only parts of the group-by arrays are needed at any time. By carefully arranging the chunk computation order, multiple cuboids can be computed simultaneously in one pass.

The MultiWay algorithm is effective when the product of the cardinalities of the dimensions are moderate. If the dimensionality is high and the data is too sparse, the method becomes infeasible because the arrays and intermediate results become too large to fit in memory. Moreover, the top-down algorithm cannot take advantage of Apriori pruning during iceberg cubing, i.e., the iceberg condition can only be used after the whole cube is computed. This is because the high-dimension to low-dimension computation order does not have the anti-monotonic property [2, 50].

BUC

BUC [9] employs a bottom-up computation by expanding dimensions. Cuboids with fewer dimensions are parents of those with more dimensions. BUC starts by reading the first dimension and partitioning it based on its distinct values. For each partition, it recursively computes the remaining dimensions. The bottom-up computation order facilitates the Apriori-based pruning: The computation along a partition terminates if its count is less than *min_sup*.

Apriori pruning reduces lots of unnecessary computation and is effective when the dataset is sparse. However, BUC does not share the computations, but sharing is very useful in computing dense datasets. We will review a closed cube method [43, 44] based on BUC later in this section.

StarCubing

Star-Cubing [72] uses a hyper-tree structure, called star-tree, to facilitate cube computation. Each level in the tree represents a dimension in the base cuboid. The algorithm takes advantages of shared computation and Apriori pruning. In the global computation order, it uses simultaneous aggregation. However, it has a sub-layer underneath based on the bottom-up model by exploring the notion of shared dimension, which enables it to partition parent group-by's and use the Apriori-based pruning on child group-by's.

Star-Cubing performs well on dense, skewed and not-so-sparse data. However, in very sparse data sets, *e.g.*, the cardinalities of dimensions are large, the star tree gets wider. It requires more time in construction and traversal. In this thesis, we first extend the original algorithm for the efficient computation in sparse data, then discuss the closed Star-Cubing method.

C-Cubing

It is well recognized that data cubing often produces huge outputs. Besides the iceberg cube, where only significant cells are kept, the closed cube is another popular solution. In closed cube, a group of cells which preserve roll-up/drill-down semantics are losslessly compressed to one cell. Due to its usability and importance, efficient computation of closed cubes still warrants a thorough study.

To efficiently compute closed cube, previous work has developed two approaches on closedness checking and pruning of non-closed cells. [73] proposed a new measure, called closedness, for efficient closed data cubing. It is shown that closedness is an algebraic measure and can be computed efficiently and incrementally. Based on closedness measure, an aggregation-based approach, called C-Cubing (*i.e.*, Closed-Cubing), is developed and integrated to Star-Cubing. The C-Cubing runs fairly efficiently.

2.2.3 High-dimensional OLAP

OLAP over high-dimensional data has been shown difficult due the curse of dimensionality. [45] proposed a new method called shell-fragment. It vertically partitions a high dimensional dataset into a set of disjoint low dimensional datasets called fragments. For each fragment, a local data cube is computed. Furthermore, the set of tuple-ids that contribute to the non-empty cells is registered in the fragment data cube. These tuple-ids are used to bridge the gap between various fragments and re-construct the corresponding cuboids upon request. These shell fragments are pre-computed off-line and are used to compute queries in an online fashion. In other words, data cubes in the original high dimensional space are dynamically assembled together via the fragments.

Data Cube has been playing an essential role in implementing fast OLAP operations [27]. The measures in the cube are generally simple statistics (*e.g.*, sum). Some recent proposals introduce more complex measures, such as linear regression model [25] and classification model [23]. This thesis discusses how to use data cube for multi-dimensional ranking analysis.

2.3 Other Related Work

A closely related study is the top-k selection queries proposed in [14], where the authors proposed to map a top-k selection query to a range query. The *soft* selection conditions in their queries are essentially the ranking functions for the k nearest neighbor search and our problem of answering top-k queries with *hard* selection conditions is not considered. We will compare ranking-cube with this method in Chapter 3.

Supporting top-k queries inside the relational query engine, in terms of a basic database functionality, has been studied by [16, 41, 45]. These approaches augment ranking into the query optimizer and therefore rank-aware query plans are considered during plan enumeration. Our work is orthogonal to them in that we focus on efficient top-k query execution in the OLAP framework.

Top-k queries are related to several other preference queries, including skyline query [12] and convex hull query [11]. Skyline query asks for the objects that are not dominated by any other object in all dimensions. A nearest-neighbor query specifies a query point p and searches for objects close to it. A convex hull is a set of points that minimizes any linear functions. The methodology developed in this thesis is also applicable to these queries, and we will discuss the extensions in this thesis.

Chapter 3

Ranking Cube

3.1 Overview

In this chapter, we propose a new computational model, called *ranking cube*, for efficient answering multi-dimensional top-k queries. We define a rank-aware measure for the cube, capturing our goal of responding to multi-dimensional ranking analysis. Using the ranking cube, we develop an efficient query algorithm. The curse of dimensionality is a well-known challenge for data cube. We cope with this difficulty (on high boolean dimensions) by introducing a new technique of ranking fragments. The solution for high ranking dimensions are more complicated, and will be discussed in Chapter 5. This chapter are organized as follows.

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

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

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

3.2 Cube Structure

In this section, we show the structure of the ranking cube and the construction method. We first consider the case where the number of boolean dimensions is small so that computing a full ranking cube over all boolean dimensions is possible. The extension to larger number of selection dimensions can be handled by *ranking fragments*, which will be presented later in this chapter.

3.2.1 General Principles

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

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

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

3.2.2 Geometry Partition

Based on the above motivation, we create a new dimension B (*i.e.*, *block dimension*) on the base table. The new block dimension organizes all tuples in R into different rank-aware blocks, according to their values on ranking dimensions. To illustrate the concept, a small database, Table 3.1, is used as a running example. Let A_1 and A_2 be categorical attributes, and N_1 and N_2 be numerical attributes.

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

Table 3.1: **An Example Database**

Suppose the block size is P. There are many ways to partition the data into multiple blocks such that (1) the expected number of tuples in each block is P, and (2) the tuples in the same block are geometrically close to each other. One possible way is the *equi-depth partition* [53] of each ranking dimension. The number of bins b for each dimension can be calculated by $b = (\frac{T}{P})^{\frac{1}{R}}$, where R is the number of ranking dimensions and T is the number of tuples in the database.

There are other partition strategies, *e.g.*, equi-width partition, multi-dimensional partition [49], *etc.*. For simplicity, we demonstrate our method using equi-depth partitioning. Our framework accepts other partitioning strategies and we will discuss this in section 3.6. Without loss of generality, we assume that the range of each ranking dimension is $[0, 1]$. We refer the partitioned blocks as *base blocks*, and the new *block dimension* B contains the *base block IDs* (simplified as *bid*) for each tuple. The original database can be decomposed into two sub-databases: the *selection database* which consists of the selection dimensions and the new dimension B, and the *base block table* which contains the ranking dimensions and the dimension B. The equi-depth partitioning also returns the meta information of the

bin boundaries on each dimension. Such meta information will be used in query processing. Example 3 shows an equi-depth partitioning of the sample database.

Example 3 *Suppose the data is partitioned into* 16 *blocks (Fig. 3.1). The original database is decomposed into two sub-databases as shown in Table 3.2. The bid is computed according to sequential orders (e.g., the four blocks on the first row are* b_1, b_2, b_3, b_4*; the four blocks on the second row are* b_5, b_6, b_7, b_8*; etc.). The selection dimension coupled with the dimension* B *will be used to compute ranking cube, and the ranking dimension table keeps the original real values. The meta information returned by the partitioning step is the bin boundaries:* $Bin_{N1} = [0, 0.4, 0.45, 0.8, 1]$ *and* $Bin_{N2} = [0, 0.2, 0.45, 0.9]$.

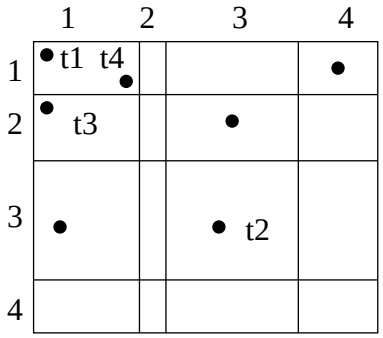


Figure 3.1: **Equi-Depth Partitioning**

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

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

Table 3.2: **Table Decomposition**

3.2.3 Rank-Aware Data Cubing

The ranking cube is built on the selection table. The measure in each cell is a list of *tid* (*i.e.*, tuple ID). A cuboid in the ranking cube is named by the involved selection dimensions

and ranking dimensions. For example, the cuboid $A_1A_2\_N_1N_2$ corresponds to selection dimensions A_1, A_2 and ranking dimensions N_1, N_2.

Using our example database, a fragment of cuboid $A_1A_2\_N_1N_2$ is shown in Table 3.3.

A_1	A_2	*bid*	*tid* List
1	1	1	1,4
1	1	5	3
1	2	11	2
...	...	...	...

Table 3.3: $A_1A_2\_N_1N_2$ **Cuboid**

Our first proposal is to organize the *tid* list with respect to the different combinations of selection dimension and the dimension B (*e.g.*, $a_1^1a_2^1b_1$, where a_1^1, a_2^1 and b_1 are values on dimension A_1, A_2 and B). Since each *bid* represents a geometry region, the materialized ranking cube is able to quickly locate the cell which is the most promising for a given top-k query. Let us call the base blocks given by the equi-depth partitioning *logical blocks* and the disk blocks used to store the *tid* list *physical blocks*. Before the multi-dimensional data cubing, each logical block corresponds to one physical block. However, with the multi-dimensional data cubing, the tuples in each logical base block (*e.g.*, b_1) are distributed into different cells (*e.g.*, $a_1^1a_2^1b_1$, $a_1^2a_2^2b_1$, *etc.*), and thus the number of tuples in each cell is much smaller than the physical block size. In order to leverage the advantage provided by block-level data access on disk, we introduce the *pseudo block* as follows. The base block size is scaled in each cuboid such that the expected number of tuples in each cell occupies a physical block. Let the cardinality of the selection dimensions of a cuboid $A_1A_2 \ldots A_S\_N_1N_2 \ldots N_R$ be $c_1, c_2, \ldots, c_s$, the expected number of tuples in each cell is $n = \frac{P}{(\prod_{j=1}^{s} c_j)}$, where P is the block size. The *scale factor* can be computed by $sf = \lfloor (\frac{T}{n})^{\frac{1}{s}} \rfloor = \lfloor (\prod_{j=1}^{s} c_j)^{\frac{1}{s}} \rfloor$. The pseudo block is created by merging every other sf bins on each dimension. We assign the *pseudo block ID* (or, *pid*) to the dimension B for each tuple, and the *bid* value is stored together with *tid* in the cell. A pseudo block partitioning is demonstrated in Example 4.

Example 4 *In Table 3.2, let the cardinalities of A_1 and A_2 be 2, and there will be 4 pseudo blocks (Fig. 3.2). The solid lines are the partitions for pseudo blocks and the dashed lines are the original partitions for base blocks. The new cuboid is shown in Table 3.4.*

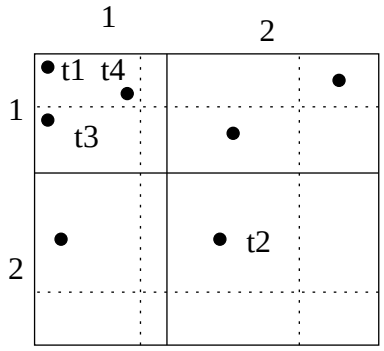


Figure 3.2: **Pseudo Block Partitioning**

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

Table 3.4: $A_1A_2\_N_1N_2$ **Cuboid After Pseudo Blocking**

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

3.3 Online Query Computation

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

3.3.1 Data Access Methods

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

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

3.3.2 Query Algorithm

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

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

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

score in the list be S_k.

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

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

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

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

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

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

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

3.3.3 A Demonstrative Example

Using the database in Table 3.1, we demonstrate each step by a running example:

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

The algorithm first determines that cuboid $C = A_1A_2\_N_1N_2$ can be used to answer the query. The related meta information is shown in Table 3.5.

Suppose the range of each ranking dimension is $[0, 1]$. The minimal value of the function $f = N_1 + N_2$ is 0 (by $N_1 = 0$ and $N_2 = 0$). The algorithm locates the first base block as b_1

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

Table 3.5: **Meta Information for answering query**

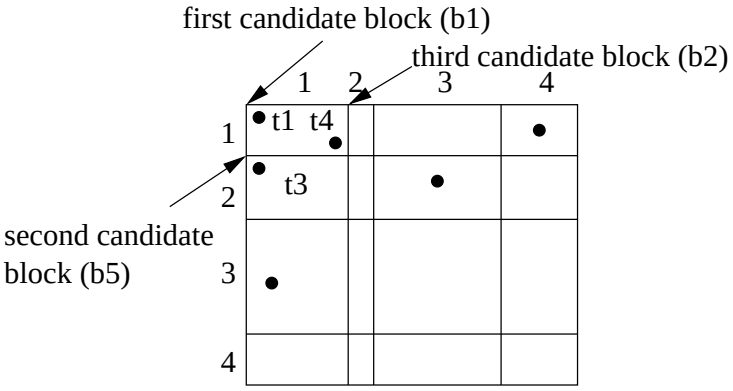


Figure 3.3: **Processing top-2 on example query**

(as shown in Fig. 3.3), and asks cuboid C to return the *tid* list of this block. The cuboid C maps b_1 to p_1 by the scale factor 2 (as shown in Fig. 3.2). Then C issues the *get pseudo block* method and retrieves the following contents: $\{t_1(b_1), t_4(b_1), t_3(b_5)\}$. t_1 and t_4 are returned as the results of the b_1 query and $t_3(b_5)$ is buffered for future queries. The algorithm then issues the *get base block* method to get the real values of tuples in b_1 and verifies that the exact score of t_1 is 0.1 and that of t_4 is 0.5. To test the stop condition, the algorithm computes the neighboring blocks of b_1. In this example, the neighboring blocks are b_2 and b_5, and thus $H = \{b_2, b_5\}$. Using the meta information on bin boundaries, we can compute the best scores of the neighboring blocks w.r.t. to the ranking function $f = N_1 + N_2$. The base block b_2 has the best score 0.4 and the base block b_5 has the best score 0.2. Both correspond to the left upper points (as shown in Fig. 3.3). Hence, $S_{unseen} = f(b_5) = 0.2$. At this stage, the list of S and the list of H are shown in Table 3.6.

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

Table 3.6: **List Values at Stage 1**

Since the current k^{th} score is $S_2 = 0.5 > 0.2 = S_{unseen}$, the stop condition is not met. The algorithm continues to pick b_5 as the next candidate block, and inserts its neighboring blocks b_6 and b_9 into H. Again, the cuboid C is asked to return the *tid* list with b_5, which is mapped to p_1. This time, the results are buffered and $\{t_3(b_5)\}$ is directly returned. The algorithm further retrieves real values for b_5 and verifies that the exact score of t_3 is 0.3. After updating the score list S, we have $S_2 = 0.3$. The list H contains blocks b_2, b_6 and b_9, and the scores associated with them are $f(b_2) = 0.4$, $f(b_6) = 0.6$, and $f(b_9) = 0.45$, respectively. Thus, $S_{unseen} = 0.4$. At this stage, the list of S and the list of H are shown in Table 3.7. Since $S_2 = 0.3 <= S_{unseen}$, the stop condition is satisfied. The algorithm returns t_1 and t_3 as top-2 results.

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

Table 3.7: **List Values at Stage 2**

3.4 Ranking Fragments

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

Before delving deeper into the *semi-online computation*, we claim the following observation about ranked query in high-dimensional space. Although a database may contain many selection dimensions, most queries are performed only on a small number of dimensions at a time. In other words, a real life ranked query is likely to ignore many selection dimensions. Stemming from the above observation, we partition the dimensions into different groups called *fragments*. The database is projected onto each fragment, and ranking cubes are fully

materialized for each fragment. With the semi-materialized fragments, one can dynamically assemble and compute any ranking cuboid cells of the original database online. In the rest of this section, we discuss the two components of our computation model: *semi-materialization* and *semi-online computation*.

3.4.1 Materializing Fragments

Here we show a general grouping framework and its storage size analysis. There are other criteria which can be exploited to group selection dimensions for efficient query processing, and we will address these issues in Section 3.6. Suppose the database has S selection dimensions and the size of the fragment is F (*i.e.*, the number of selection dimensions in the fragment), we evenly partition the selection dimensions into $\frac{S}{F}$ disjoint sets. Each fragment will combine with the ranking dimensions to construct a ranking cube. An example of fragment grouping is shown in Example 5.

Example 5 *Suppose a relation has* 4 *selection dimensions* $A_1, \ldots, A_4$ *and two ranking dimensions* N_1, N_2. *We evenly group the selection dimensions into two fragments* (A_1, A_2) *and* (A_3, A_4) *and the ranking fragments are:* (A_1, A_2, N_1, N_2) *and* (A_3, A_4, N_1, N_2).

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

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

$1) + (R + 2)T)$.

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

3.4.2 Answering Query by Fragments

Given the semi-materialized ranking fragments, one can answer a ranked query on the original data space. We say a cuboid *covers* a query if all the dimensions involved in the query appear on the materialized cuboid. In this case, the query can be directly answered using the query algorithm described in Section 3.3. Otherwise, the query is answered by a set of cuboids which, as a whole, cover the query.

Determining Covering Cuboids

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

Example 6 *Suppose the materialized fragments are* $(A_1,\ A_2, N_1, N_2)$ *and* (A_3, A_4, N_1, N_2). *The query consists of the selection dimensions* $(A_1,\ A_4)$ *and ranking dimensions* $(N_1,\ N_2)$.

We first locate the set of candidate cuboids $MD = \{A_1\_N_1N_2,\ A_4\_N_1N_2\}$*, and this is also the minimum covering subset* MS*.*

Computing Cuboid Cells Online

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

Suppose the query has the selection dimensions (A_1, A_4). The fragments demonstrated in Example 5 are used and a covering set of two cuboids $A_1\_N_1N_2$ and $A_4\_N_1N_2$ is selected. Our goal is to online compute the required cells of cuboid $A_1A_4\_N_1N_2$. Based on the query algorithm presented in Section 3.3, we only need to make a small change at *retrieve* step. Instead of issuing the *get pseudo block* method to cuboid $A_1A_4\_N_1N_2$, which was not materialized, we issue the *get pseudo block* method to cuboid $A_1\_N_1N_2$ and $A_4\_N_1N_2$. The *tid* lists returned by both cuboids will be *intersected* as the answer. All the other steps in the query algorithm (Section 3.3) remain the same. The *merge and intersect* operation on the tif lists can be generalized to more than two ranking fragments.

3.5 Performance Study

This section reports the experimental results. We compare the query performance of ranking cube and ranking fragments with two other alternatives: the *baseline* solution by Microsoft SQL-Server and the *rank mapping* approach proposed by [14]. We first discuss the experimental settings, and then show the results on low dimensional data (with ranking cube) and high dimensional data (with ranking fragments).

3.5.1 Experimental Setting

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

Data Sets

We use both synthetic and real data sets for the experiments. The real data set we consider is the *Forest CoverType* data set obtained from the UCI machine learning repository web-site (www.ics.uci.edu/∼mlearn). This data set contains 581, 012 data points with 54 attributes, including 10 quantitative variables, 4 binary wilderness areas and 40 binary soil type variables. We select 3 quantitative attributes (with cardinalities 1, 989, 5, 787 and 5, 827) as ranking dimensions, and other 12 attributes (with cardinalities 255, 207, 185, 67, 7, 2, 2, 2, 2, 2, 2, 2) as selection dimensions. To achieve a reasonable size of the data, we further duplicate the original data set 5 times and the data set has 3, 486, 072 tuples. We also generate a number of synthetic data sets for our experiments. The parameters and default values are summarized in Table 3.8.

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

Table 3.8: **Parameters for Synthetic Data Sets**

Experimental Configurations

In the experiments, we compare our proposed approach against the baseline solution provided by commercial database and the rank mapping technique discussed in [14]. All the experiments are conducted over Microsoft SQL Server 2005 on a 3GHz Pentium IV PC with 1.5GBytes of RAM. Specifically, we use the following techniques for answering top-k queries.

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

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

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

Rank Mapping Approach: In [14], the authors proposed to map a top-k selection query to a range query. Their problem definition is slightly different from ours since the top-k queries in [14] do not have *hard* selection condition. However, the idea of mapping a ranking function to a range query can also be applied in our problem. We refer their method as *rank mapping*. An example of applying rank mapping in our problem is as follows:

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

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

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

the multi-dimensional range query.

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

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

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

Evaluation Metric

We use *execution time* to evaluate the techniques presented above. For each experiment, we report the average running time for executing a set of 20 randomly issued queries. Without loss of generality, we use linear ranking functions in our evaluation. One criterion to measure the query difficulties of the linear ranking functions is the query skewness, which is defined

as follows. For a linear ranking function $\alpha_1 N_1 + \alpha_2 N_2 + \ldots + \alpha_r N_r$, let $\underline{\alpha} = \min_{i=1}^{r} \alpha_i$ and $\overline{\alpha} = \max_{i=1}^{r} \alpha_i$, the query skewness is defined as $u = \overline{\alpha}/\underline{\alpha}$.

The parameters and their default values for queries are shown in Table 3.9.

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

Table 3.9: **Parameters for Queries**

3.5.2 Experiments on Ranking Cube

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

Top-k Query: Figure 3.4 reports the execution time as a function of k (*i.e.*, the number of tuples requested) on the default synthetic data. Our methods is much more efficient than the previous approaches. As expected, the baseline approach is not sensitive to the value of k since it retrieves all the tuples. The execution time of the rank mapping approach increases slightly. This is because we assign the optimal bound values for range queries and those bound values increase slightly. Our method progressively retrieves data blocks, and thus a larger k value asks for more data accesses. When $k = 10$, the ranking cube is 4 times faster than the rank mapping approach and 10 times faster than the baseline approach.

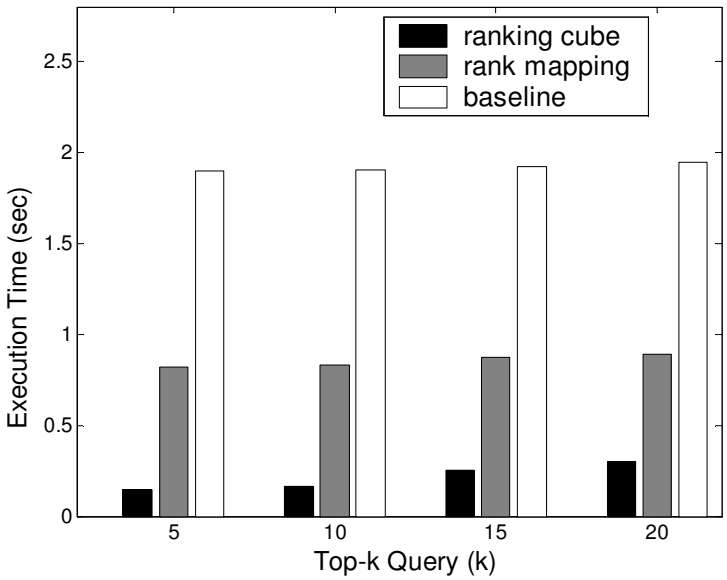


Figure 3.4: **Query Execution Time w.r.t.** k

Query Skewness: In this experiment, we measure the performance by varying query skewness. The goal of this experiment is to test the robustness of the base block partitioning. A skewed query may require the system to retrieve more data since the top results may be distributed on more base blocks. We vary the value of u and the results are reported in Figure 3.5. The execution time of our method increases slightly with u. However, it still performs much better comparing with other alternatives.

Number of Dimensions in Ranking Function: Here we continue to test the query performance with respect to the ranking function. We generate a synthetic data with 3 selection dimensions and 4 ranking dimensions, and vary the value of r (the number of dimensions involved in the ranking function) from 2 to 4. The results are shown in Figure 3.6. We observe that the execution time of our method slightly increases when the number of dimensions decreases. This is because our base block table is constructed on all 4 ranking dimensions. A 2 dimensional query means the blocks need to be projected onto the lower dimensions. Hence more block accesses are required. The baseline approach is not sensitive to r. The rank mapping approach performs worse because the bound estimation, although it is optimal, is much looser in higher dimensions.

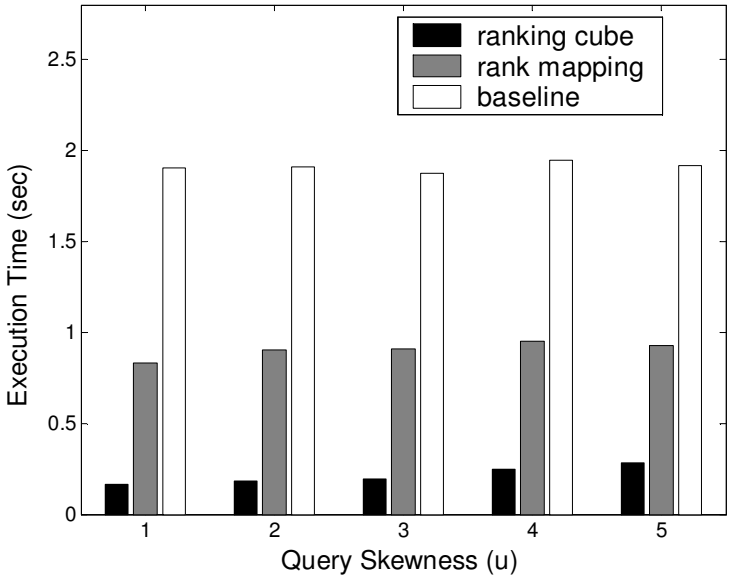


Figure 3.5: **Query Execution Time w.r.t.** u

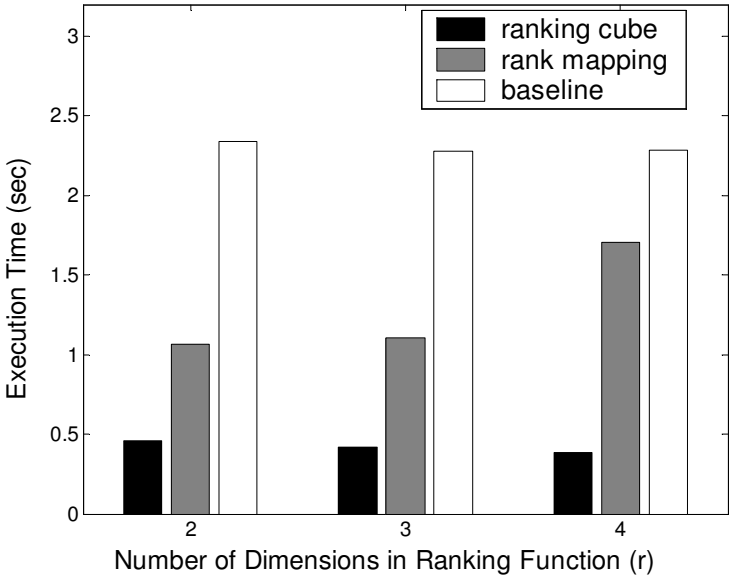


Figure 3.6: **Query Execution Times w.r.t.** r

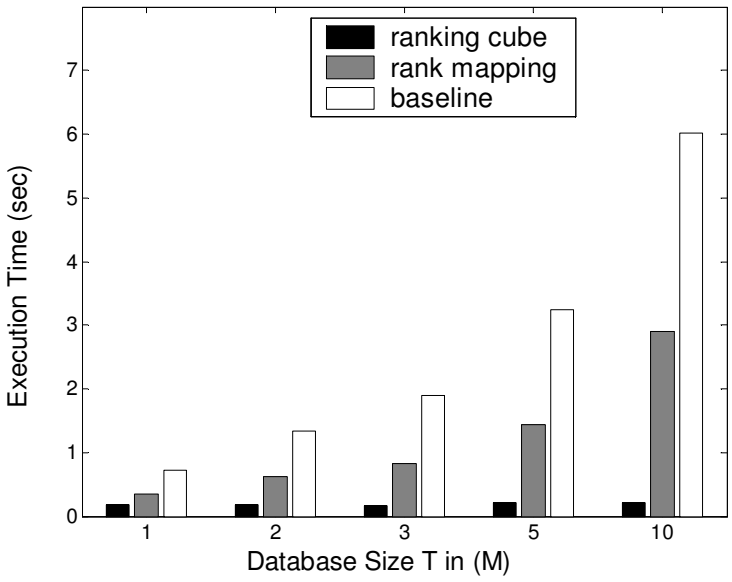


Figure 3.7: **Query Execution Time w.r.t.** T

Database Size: To analysis the query performance with respect to the data size, we change the number of tuples in the synthetic data from $1M$ to $10M$ (Figure 3.7). The baseline performs worse on larger data set since the selection conditions return more qualified tuples and the database needs to do more random accesses. Although the rank mapping approach is fed by the optimal bound values, it performs worse with larger data size. This may be caused by the reason that query execution time is sensitive to the dimensions involved in the query. If the involved dimensions exactly follow the order on which the multi-dimensional index was built, it can be answered extremely fast. Otherwise, there will be more random access and the execution time is also affected by the data size. On the other hand, the ranking cube approach has stable performance, regardless of the data size. This indicates that our proposed approach is especially attractive for larger data set.

Cardinality: We generate a set of synthetic data by varying the cardinality of each selection dimension from 10 to 100. The results are shown in Figure 3.8. Basically, increasing cardinality favors the baseline approach since the number of tuples filtered by the selection conditions decreases significantly. There is no clear trend of the rank mapping approach in this experiment, mainly because we assign the optimal bound value for the transformed

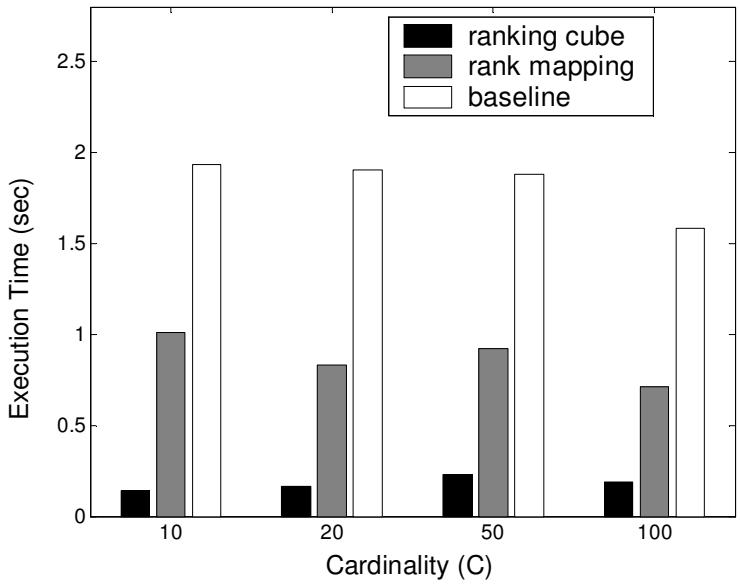


Figure 3.8: **Query Execution Time w.r.t.** C

range query and the number of tuples satisfying the range query is thus not sensitive to the cardinalities. For our method, the scale factor (see Section 3.2) of the pseudo block increases with the cardinality and the tuples are more sparse in each pseudo block. As the result, it invokes more *get base block* methods to verify the tuples and the execution time slightly increases. When C is large enough (*i.e.*, 100), we observe the execution time decreases again. This is because the number of tuples retained by the selection conditions is quite small and many base blocks are not retrieved since they are found to be empty during the *get pseudo block* step. This is consistent with our analysis that the combination of two block access methods is robust (see Section 3.3.1).

Number of Selection Conditions: Here we vary the number of selection conditions to test the trade-offs between the ranking and selection. The results are shown in Figure 3.9. Generally, involving more selection conditions results in fewer qualified tuples, and consequently, the baseline approach improves its query performance. The execution time of rank mapping decreases because the multi-dimensional index has better utilization. In the experiment, we use a synthetic data with 4 selection dimensions. When the dimensions involved in the query are the same as those in the multi-dimensional index (*i.e.*, $s = 4$ in

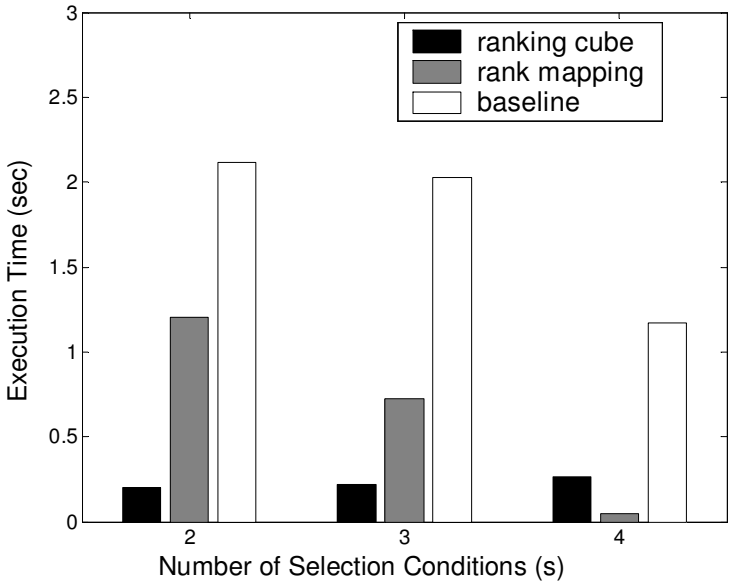


Figure 3.9: **Query Execution Time w.r.t.** s

Figure 3.9), the range query can be answered very fast. The execution time of ranking cube slightly increases with the number of selection conditions, and overall its performance is not sensitive to the number of selection conditions, for the same reason given in the previous experiment. We also observe that with 4 selection conditions, the number of qualified tuples is 22. Ranking is even not necessary in this case.

Block Size: The final experiment with ranking cube is to test the sensitivity with respect to the block size B. We measure the block size by the expected number of tuples contained by a block. The results are shown in Figure 3.10. We observe that the execution time is within 10% between each other cases, and the performance not sensitive to the value of B (from 100 to 1000). In our experiments, we use 300 as the default value for B.

3.5.3 Experiments on Ranking Fragments

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

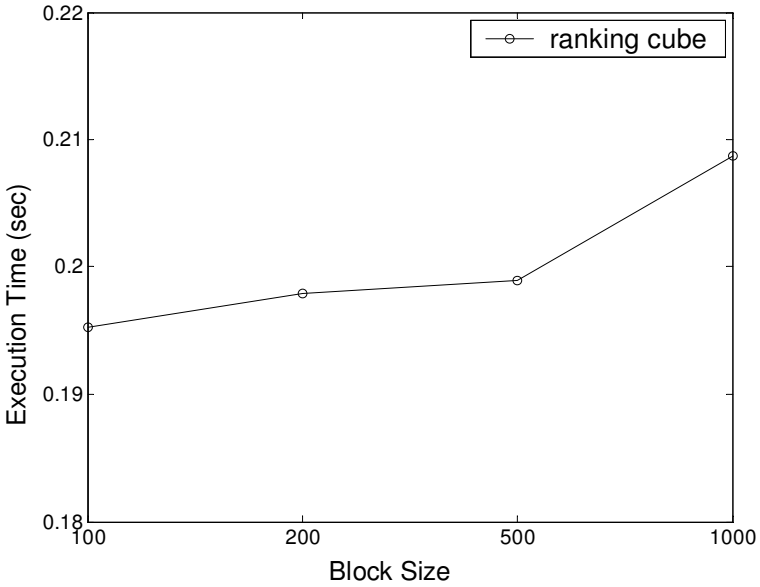


Figure 3.10: **Query Execution Time w.r.t. Block Size**

number of selection conditions to 3.

Space Consumption: The first experiment is to examine the amount of space needed to store the ranking fragments. Specifically, how it scales as the dimensionality grows. We use a synthetic data with 3-12 selection dimensions and build ranking fragments with $F = 2$. Figure 3.11 shows the total space consumption in SQL server. The space usage includes both the data and the indices. We compare the total space usage with baseline (BL) and rank mapping (RM) approaches. The baseline approach builds a non-clustered index on each selection dimension and the rank mapping approach builds a multi-dimensional index for each ranking fragment. Although neither of them generates new tables, the indices used by them are much larger than the the base table. We also observe that the clustered indices built by the ranking fragments (RF) occupy small space (roughly 1% of the total space). As shown in the figure, the space usage of all three methods grow linearly with the number of dimensions. The space used by ranking fragments is only 2-2.5 times of that of the other two alternatives. It is a fairly acceptable cost paid for materialization since the online query processing becomes much more efficient. Comparing with other data cube proposals, the space requirement by the ranking fragments is more practical. Furthermore, since we store

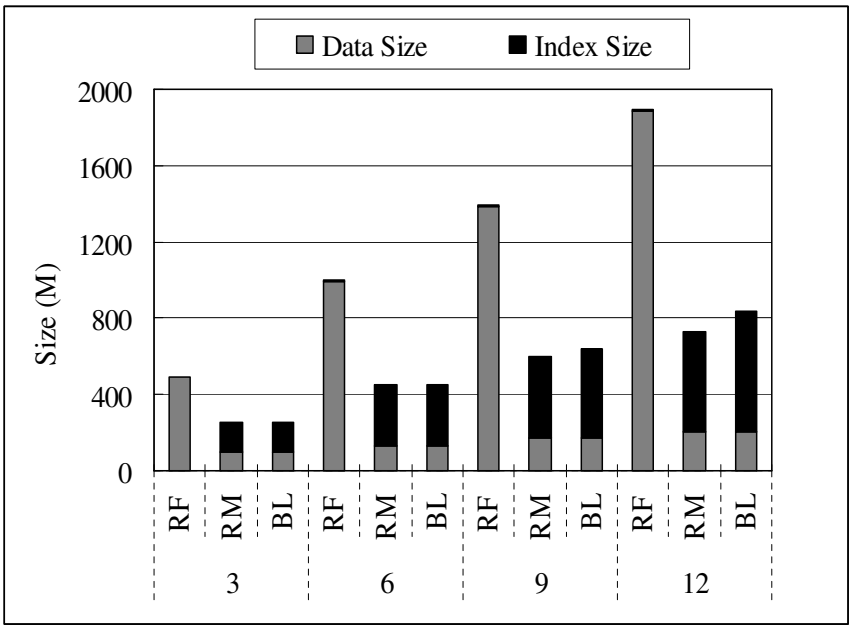


Figure 3.11: **Space Usage w.r.t. Number of Selection Dimensions**

ranking fragments in relational database, a large portion of the space is used to store the cell identifiers. We believe that the space requirement can be further reduced if we store the data out of the relational database. We discuss more compression opportunities in Section 3.6.

Number of Covering Fragments: Since the ranking fragments do not guarantee to cover a query by a single fragment, we test the query performance with respect to the number of covering fragments. We generate 3 top-k queries, each involving 3 selection dimensions, and intentionally let them be covered by one, two and three fragments. We use the same data set as described above, and the execution time is shown in Figure 3.12. The execution time increases with the number of covering fragments. Typically, the execution time with 2 (3) covering fragments is roughly 1.4 (2) times of that with one covering fragment. We observe that even with 3 covering fragments, our method is still around 4 times faster than the baseline and 2 times faster than the rank mapping approach (See Figure 3.4).

Fragment Size: Here we test the query performance with respect to the fragment size. A larger fragment size will have better coverage of the queries, however, it requires more

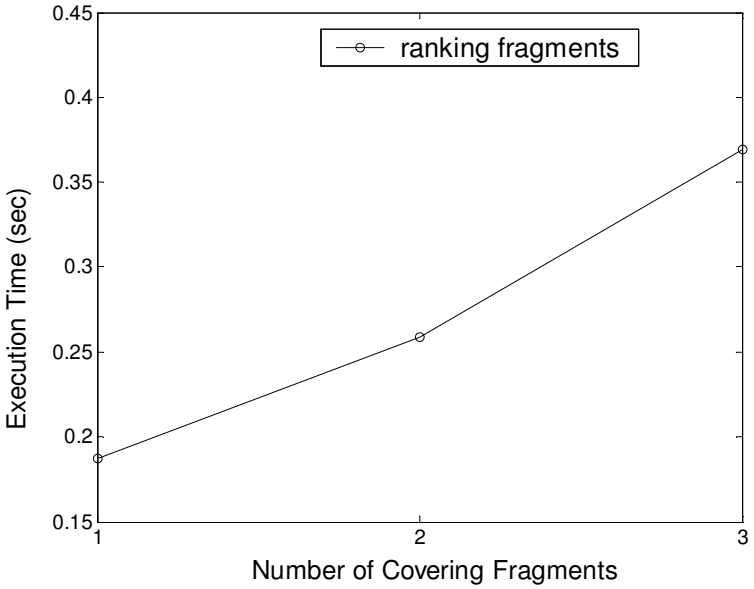


Figure 3.12: **Query Execution Time w.r.t. Number of Covering Fragments**

space usage as well. Generally, fragment size larger than 3 is too space consuming, and we test the performance on fragment sizes 1, 2, and 3. We continue to use the same synthetic data set described above and the queries are generated with 3 selection conditions. The results are shown in Figure 3.13. We observe that the larger fragment size provides better query performance.

Number of Dimensions: Here we fixed the fragment size as 2 and vary the number of selection dimensions from 3 to 12. We compare the query execution time with the baseline and the rank mapping approaches. The results are shown in Figure 3.14. We have the following observations. First, the baseline approach is not sensitive to the number of dimensions. Second, the rank mapping approach becomes worse when dimension increases. This is because in high dimensional data, the multi-dimensional index in each fragment has low probability to cover a query. In many cases, the query only accesses one dimension in a multi-dimensional index and this is quite expensive. Finally, the time used by ranking fragments increases slightly from 3 to 9 dimensions. For queries with three selection conditions, we expect that the execution time will keep stable with higher dimensions. This is because query execution in the worst case (*i.e.*, the query is covered by three different fragments) is

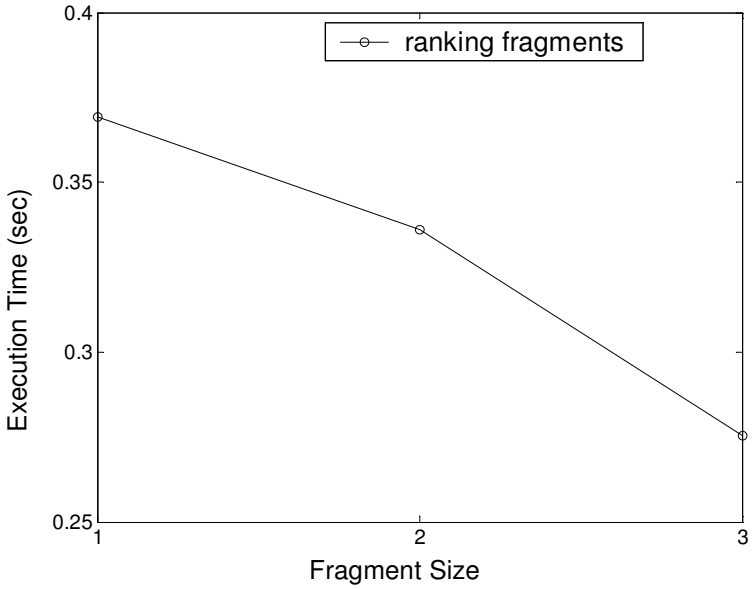


Figure 3.13: **Query Execution Time w.r.t. Fragment Size**

still quite efficient (see Figure 3.12).

Real Data: Besides synthetic data, we also tested our proposed methods on the real-world data set: Forest CoverType (see Section 3.5.1). We evenly partition the 12 selection dimension into 4 groups (*i.e.*, fragment size is 3). The queries has 3 selection conditions and the ranking function spans on all three ranking dimensions. The query execution time with respect to k is shown in Figure 3.15. Different from the result in last experiment, here we observe that the rank mapping approach is more efficient than the baseline approach. This is because in this real data set, many dimensions have cardinality 2. As a result, the baseline approach needs to access more tuples. The low cardinalities also enable the rank mapping approach to efficiently access the multi-dimensional indices. Comparing with these two alternatives, our proposed method consistently performs the best on both the synthetic and read data sets.

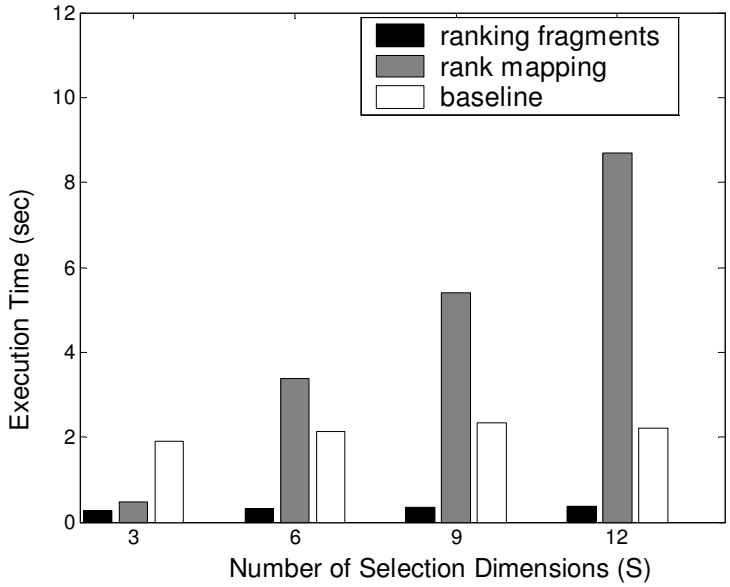


Figure 3.14: **Query Execution Time w.r.t.** S

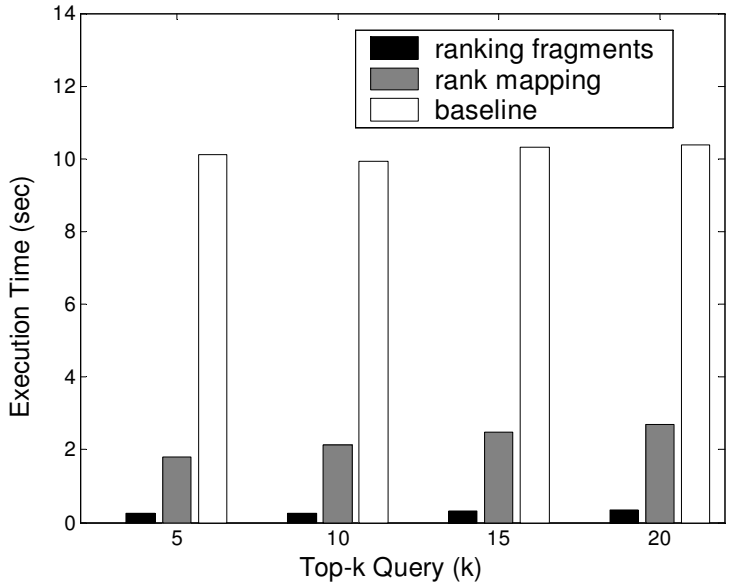


Figure 3.15: **Query Execution Time on Real Data**

3.6 Discussion

To efficient process top-k queries with multi-dimensional selections, we proposed a novel rank-aware cube structure which is capable to simultaneously handle ranked queries and multi-dimensional selections. Based on the ranking cube, we develop a progressive query processing algorithm. We further extend the ranking cube to ranking fragments, which is especially useful for high dimensional data. Here we discuss some possible extensions.

3.6.1 Ad Hoc Ranking Functions

Although we demonstrated the solution by convex function, the framework can be extended to *ad hoc* ranking functions. The basic idea is to decompose the whole domain of the function variables into multiple sub-domains so that in each sub-domain, the function has convex property. The decomposition relies on the scientific computing techniques and is out of the scope of this thesis. After the convex sub-domains are computed, we can fetch those starting points in each sub-domain. The algorithm can be modified to merge the current best tuples from each sub-domain and maintain a global neighboring block list (See Section 3.3), which determines the next candidate block for retrieving tuples.

3.6.2 Variations of Ranking Cube

We have used equi-depth partitioning to build the ranking cube. The proposed methods can also be combined with other partitioning strategies. For example, a multi-dimensional partitioning [49] recursively partitions the data domain, one dimension at a time, into bins enclosing the same number of tuples. Each multi-dimensional bin can be considered as a base block. We can still use the concept of pseudo block by merging sf_i number of consecutive base blocks in each dimension i, where sf_i is the scale factor (See Section 3.2) of dimension i. The query algorithm remains the same. The trade-off is that we need more space for the meta information of the base block partitioning.

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

In this thesis, we assume the boolean predicates come from categorical attributes. We can handle ranking cubes over numerical attributes in two ways. First, the domain of numerical attributes can be discretized and the ranking cubes are then built based on the discretized values. During query evaluation, the boolean predicates can be relaxed to the discretization granularity. A final checking step is needed to verify the exact numerical values. Secondly, one can build a B-tree or R-tree partition over the numerical attributes, and handle the boolean predicates B as a special ranking function such that if a tuple t does not satisfy B, the value of $f(t)$ is infinite large (for minimal query).

3.6.3 ID List Compression

The *tid* list in each block can be compressed, such that each block contains more *tid*s. As the result, the system will retrieve less number of blocks for evaluating a ranked query. One compression method is the bitmap indexing [3, 18]. In many applications, the cardinalities of selection dimensions are small. For example, in the used car database, the majority of selection dimensions only have 2 possible values, *e.g.*, whether it has power window, sunroof, and so on. The bitmap indexing can be used to compress the *tid* lists in the ranking cube and improve the space usage. Furthermore, the merge operation in ranking fragments can be performed much faster using the bit-AND operation than the standard merge-intersect operation.

Another compression method of the *tid*-lists come from information retrieval [60]. The

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

3.6.4 High Ranking Dimensions

In this chapter, we assume that the number of ranking dimensions is not large. In some applications where more ranking dimensions are involved, we can construct a variant of ranking fragments. Similar to our proposal on handling high selection dimensions, we can partition the ranking dimensions into several groups. The ranking fragments can be assembled by picking one group from selection dimensions and one group from ranking dimensions. If a query falls into two ranking fragments with different ranking groups, the query processing algorithm can be extended as follows. First, the starting base blocks are found in each ranking fragments and the *tid* lists are merged as we did in Section 3.4. The system then examine the neighboring blocks in each fragment and compute the best combination of the neighboring blocks as the next candidate blocks. This procedure repeats until the stop condition is satisfied. We will discuss this in detail in Chapter 5 and Chapter 6.

Chapter 4

Signature as Measure

4.1 Overview

In last chapter, each ranking dimension is partitioned in equi-depth bins, and data tuples are grouped into grid cells by intersecting the bins from all participating preference dimensions. For each group-by, the cube stores $\langle cell\_id, tuple\_id\_list \rangle$ pairs for each cell. During top-k query execution, the online search algorithm first locates the cell which contains the extreme point, and then progressively expands to neighboring cells. This search method is confined to grid partition and convex functions. The progressive expansion requires the neighbor cells to be well defined, which is not the case in some other data partition methods (*e.g.*, R-tree). Also, relying on grid partition makes the query performance sensitive to data distribution, since the search algorithm may request many "dead" (*i.e.*, empty) cells. In this chapter, we discuss an alternative ranking cube implementation based on the hierarchical partition method, and works with a more general branch-and-bound search framework. We also exploit data compression to make the ranking cube more compact.

4.1.1 A Unified Framework

OLAPing ranked queries is essentially a task to efficiently find top answers (according to a function f) with a set of multi-dimensional boolean predicates, B. To do this, an algorithm can first filter data tuples by B and then compute the top-k results, or first search data tuples according to f and verify B on each candidate. Both approaches may retrieve data

that will be pruned by the other criterion later, and thus are not efficient. Ranking cube is a way to simultaneously combine both ranking and boolean pruning. The whole framework consists of three components.

1. To facilitate rank-aware data retrieval, data is partitioned into n blocks according to ranking dimensions. We refer the data partition as $P = \{b_1, b_2, \ldots, b_n\}$, where $b_i = \{t_i^1, t_i^2, \ldots, t_i^l\}$ is the i^{th} data block, and t_i^j is the j^{th} tuple in b_i.

2. For each B, we compute a measure $M(P|B) = \{m(b_1|B), m(b_2|B), \ldots, m(b_n|B)\}$, where $m(b_i|B) = \{\delta(t_i^1|B), \delta(t_i^2|B), \ldots, \delta(t_i^l|B)\}$ and $\delta(t_i^j|B) = 1$ if the tuple t_i^j satisfies B. The ranking cube C pre-computes and stores the measure M for all possible dimensional values.

3. Guided by C, the query processing algorithm S searches for top answers over P such that a block b_i is retrieved if and only if b_i may contain tuples better than the current top-k results, and $m(b_i|B) \neq 0$.

4.1.2 Implementation Issues

In the framework presented above, the data partition P and ranking cube C contain the semi off-line materialization, and the search algorithm S conducts the semi online computation. In this subsection, we cast two typical implementations in this framework: the *grid partition with neighborhood search* and the *hierarchical partition with top-down search*. In each implementation, we will discuss the partition scheme, the measure composition, and the query algorithm.

Grid Partition

Partition Scheme: We demonstrate the grid partition by the method used in [70]. For each ranking dimension (*e.g.*, X and Y in Table 4.1), we partition the domain into L bins by equi-depth partitioning. In our sample database (Table 4.1), suppose A and B are boolean

dimensions, and X and Y are ranking dimensions. Each ranking dimension is partitioned into 4 bins, and the data is partitioned into 16 blocks (Figure 3.1). We store the tuples with the values on ranking dimensions, cell by cell, in a *block table*. The bin boundaries of the equi-depth partition are stored in the *meta table*.

Measure Composition: Following the above example, suppose the block on the i^{th} row and j^{th} column in Figure 3.1 is b_{4i+j} $(i, j = 0, 1, 2, 3)$. Given a boolean predicate $B = a_1 b_1$, only $b_1 = \{t_3\}$ and $b_4 = \{t_1, t_2\}$ contain tuples satisfying B (*i.e.*, t_1, t_3). Consequently, $m(b_1|B) = \{1\}$, $m(b_4|B) = \{10\}$ and $m(b_i|B) = 0$ for all the other b_i. We point out two important issues in implementing $M(P|B)$. First, by merging all $m(b_i|B)$, $M(P|B)$ is basically a bit-array and can be compressed by many compression methods. Secondly, $M(P|B)$ can be decomposed into several smaller parts (while preserving neighboring b_i together), each of which is retrieved from the ranking cube only when necessary.

Query Algorithm: Given a ranked query with f and B, the query algorithm searches for blocks that are (1) promising with respect to f, and (2) containing tuples satisfying B. To begin with, we define $f(b_i)$ as the minimal value of f over the region covered by block b_i(In this thesis, we assume minimal top-k is requested.). Given the bin boundaries stored in the *meta table*, the search algorithm is able to sort all b_i according to $f(b_i)$, and retrieves b_i one by one. The algorithm skips a b_i if $m(b_i|B) = 0$ since it contains no tuple satisfying B. To avoid enumerating all blocks, one may first locate the blocks that contain the extreme points, and progressively search over their neighboring blocks. Among the neighboring blocks, the algorithm will first examine the block with minimal value of $f(b_i)$. We refer this search method as *neighborhood search*, and it assumes that the ranking functions are *convex*.

Hierarchical Partition

Partition Scheme: For hierarchical partition, we use R-Tree as an example [36]. R-Tree splits space with hierarchically nested and possibly overlapping boxes. Each node stores the pointers to child nodes and the bounding box of child nodes. The leaf node stores the *tid*s

and the values on ranking dimensions. Figure 4.1 shows a sample R-Tree, where the root node contains pointers to child nodes N_1 and N_2, and so on.

Measure Composition: Each node N_i in the R-Tree partition corresponds to a block in the ranking cube framework, and we define $m(N_i|B) = \{\delta(n_i^1|B), \delta(n_i^2|B), \ldots, \delta(n_i^l|B)\}$, where n_i^j is a child node or a tuple. $\delta(n_i^j|B) = 1$ if and only if n_i^j contains a tuple satisfying B.

Query Algorithm: Given the hierarchical partition, the algorithm follows the branch-and-bound principle to progressively retrieve data nodes. Specially, the search process first inserts the root node into a heap h. At each step, the algorithm fetches the node N appearing at the top of h (*i.e.*, with minimal value of $f(N)$), and inserts all child nodes n_j of N to h if $m(n_j|B) \neq 0$. The query processing halts when $f(N)$ is no less than the current top-k results. We refer this search method as *top-down search*.

Comments on Partition Schemes

We have presented a general framework for ranking cube, and demonstrated it by both grid and hierarchical partitions. The grid partition is simple and easy to implement. However, the query performance may be sensitive to data distribution, since the there may be many dead (i.e., empty) cells for skewed data. The hierarchical partition is more robust with respect to data distribution. But it may incur additional cost to build and traversal over the partition. In real application, one may choose different partition scheme accordingly.

4.1.3 Query Model

Without losing generality, we assume that users prefer *minimal* values. The query results are a set of objects that belong to the data set satisfying the boolean predicates, and are also ranked high (for top-k) or not dominated by any other objects (for skylines) in the same set. For top-k queries, we assume that the ranking function f has the following property: *Given a function $f(N'_1, N'_2, \ldots, N'_j)$ and the domain region Ω on its variables, the lower bound of f over Ω can be derived.* For many continuous functions, this can be achieved by computing

the derivatives of f.

4.2 Signature based Materialization

We compute a signature as the measure for the ranking-cube. The signature uses one bit (*i.e.*, 0/1) to indicate whether a data partition contains tuples satisfying the boolean predicates. For efficient storage, retrieval, and incremental maintenance, the signatures are further decomposed and compressed. In the rest of this chapter, we first describe how to construct signatures. This includes signature generation, compression and decomposition. We then discuss how to incrementally maintain signatures.

4.2.1 Signature Generation

Signatures are generated by a partition scheme and group-by conditions. We demonstrate our methods using a sample database (Table 4.1), where A and B are boolean dimensions, X and Y are preference dimensions, and *path* (see Section 4.2.1) is generated by our algorithm to facilitate the computation of ranking-cube.

tid	A	B	X	Y	path
t1	a1	b1	0.00	0.40	$\langle 1,1,1\rangle$
t2	a2	b2	0.20	0.60	$\langle 1,1,2\rangle$
t3	a1	b1	0.30	0.70	$\langle 1,2,1\rangle$
t4	a3	b3	0.50	0.40	$\langle 1,2,2\rangle$
t5	a4	b1	0.60	0.00	$\langle 2,1,1\rangle$
t6	a2	b3	0.72	0.30	$\langle 2,1,2\rangle$
t7	a4	b2	0.72	0.36	$\langle 2,2,1\rangle$
t8	a3	b3	0.85	0.62	$\langle 2,2,2\rangle$

Table 4.1: **A Sample Database** R

Partitioning Data as Template

Data is partitioned according to the ranking dimensions. There are extensive studies on different data partitioning methods. In this chapter, we use R-tree as an example. The

same concept can be applied with other multi-dimensional partition methods. We assume the data partition always has hierarchical property. Some partition methods (*e.g.*, grid partition in the last chapter) may not generate tree-structure initially. In this case, we can create hierarchies by iteratively merging neighboring grid cells. R-Tree splits space with hierarchically nested and possibly overlapping boxes. Each node of an R-tree has a variable number of entries (between the minimal value m and the maximal value M [36, 8]). Each entry within a non-leaf node stores two pieces of data: pointers to child nodes and the bounding box of child nodes. Each entry within a leaf node stores the *tid* and the values on ranking dimensions. The R-tree partition of the sample database is shown in Figure 4.1, where the root node contains pointers to child nodes N_1 and N_2, and node N_1 contains pointers to child nodes N_3 and N_4, and so on.

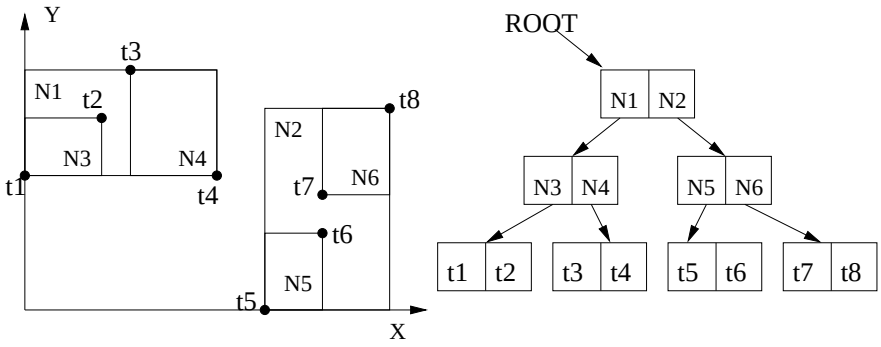


Figure 4.1: **Partition data by R-tree,** $m = 1, M = 2$

The grid partition presented in the last chapter does not have hierarchical properties. We can create a tree structure as follows. Suppose the node of the tree can hold maximum M child entries, and the number of numerical attributes is n. We evenly divide neighboring bins on each attributes into $\lfloor \log_n M \rfloor$ larger bins, which leads to $(\lfloor \log_n M \rfloor)^n \leq M$ larger grid cells. Those cells are the first level nodes linked to the root. Using the same method, we recursively partition those large grid cells until all leaves are base blocks. A sample tree construction is shown in Figure 4.2, where the upper-left four base blocks are merged to a higher level grid cell $N1$. Empty cells are removed from the tree.

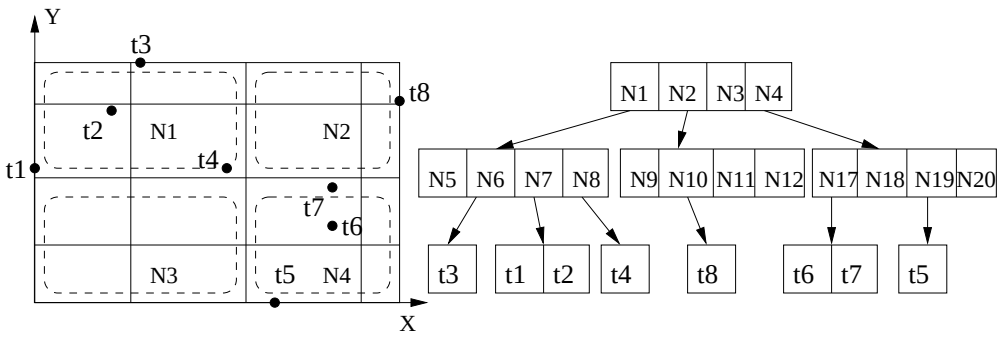


Figure 4.2: **Partition data by grids,** $k = 4, n = 2, M = 4$

Generating Signatures for Group-bys

Given a hierarchical structure of the partition scheme, we generate one signature for each cell (*e.g.*, $A = a_1$) as follows. Each node in the R-tree contains a list of pointers to its child nodes, and the length of the list is up bounded by M. We encode the list using a bit array, where each bit corresponds to a child node. If there is no data belonging to the cell (*e.g.*, $A = a_1$) in a child node (including all its descendant nodes), we set the bit value as 0. Otherwise, the bit value is 1. As a result, the signature also has a tree structure. Figure 4.3.a shows an example signature for the cell ($A = a_1$), based on the partition scheme shown in Figure 4.1.

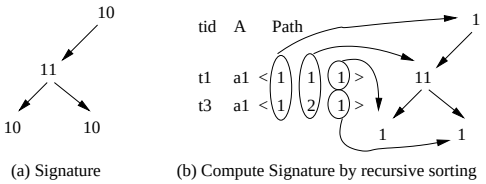


Figure 4.3: $(A = a_1)$**-signature**

A data cube may consist of many cells. A naïve method to generate a signature for each cell is to traverse the R-tree and verify on every tuple t whether t belongs to the cell. This is obviously not a scalable solution. Alternatively, we can compute all signatures for a cuboid in a tuple-oriented way. Note that every tuple is associated with a unique path from the root. The path consists of a sequence of pointer positions: $\langle p_0, p_1, \ldots, p_d \rangle$ ($1 \leq p_i \leq M$, for all i), where level-d corresponds to a leaf. For example, the path of the tuple t_1 is $\langle 1, 1, 1 \rangle$,

and that of the tuple t_3 is $\langle 1, 2, 1\rangle$.

To compute all signatures in a cuboid (*e.g.*, cuboid A), we sort tuples according to the values on dimension A, and extract sub-lists of tuples which share the same value on dimension A. For example, Figure 4.3.b shows a sub-list of tuples with $A = a_1$. For each sub-list, we compute a signature as follows: (1) sort the tuples according to p_0, (2) scan the sorted list, and set those p_0 bits to 1 in the root bit-array of the signature, and (3) for each sub-sub-list of tuples that share the same value of p_0, sort it on p_1, and insert the distinct p_1 to the first child node (*i.e.*, another bit-array) of the root. This procedure is recursively called until the whole signature is completely built. An example is shown in Figure 4.3.b.

There are many developed algorithms on efficient data cubing, and some of them can be directly plugged into our framework. For example, it is not difficult to show that the signature measure is *distributive*, and that enables shared computation cross multi-dimensional cuboids [72, 71]. If the database size is too large to fit in memory, database partitioning techniques can be applied [48].

Besides tuples, we can also assign a path for a node, such that an l-level node corresponds to a path $\langle p_0, p_1, \ldots, p_{l-1}\rangle$. In the future, we will use the path to reference a node in the signature. For simplicity, we one-to-one map a path to a *signature ID* (*e.g.*, SID) as $SID = p_0 \times (M+1)^l + p_1 \times (M+1)^{l-1} + \ldots + p_{l-1}$. In our example, $M = 2$ and the path of the node N_3 is $\langle 1, 1\rangle$. Its SID is 4.

We have the following observations on the signature. First, a signature is a tree, whose nodes contain bit arrays; and second, 0 bit corresponds to leave (*i.e.*, no sub-trees under the bit). For storage purpose, we will rewrite the tree-based signature as a binary string.

From Tree-structure to String

For storage purpose, we will rewrite the tree-based signature as a binary string. This can be done by sequentially recording the bit arrays in nodes by a bread-first traversal. In order to recover the original tree structure from the binary string, it is important to extract each node

from the binary string unambiguously. Here we describe a baseline (**BL**) coding scheme, and more advanced coding schemes are discussed in the next subsection.

A node may contain $b \leq M$ children. **BL** first uses $\lceil \log_2 M \rceil$ bits to encode the value of b, and then write the actual bit array afterwards. To save the space, we apply the *one-less principle*: Suppose the actual length is b, we code $b-1$ in the $\lceil \log_2 M \rceil$ bits. For example, the bread-first traversal coding of Figure 4.3.b is 01111010. To recover the tree structure from this bit array, one reads the first bit 0 (*i.e.*, $\lceil \log_2 M \rceil = 1$), indicating that the following 1 bit forms a node. 1 is extracted as the root, and the code implies that there is no right child. Consequently, we extract the next node 11. This is the left child of the root.

Although a signature is a compact description, the costs of storing and retrieving a complete signature may become relatively high in large databases. In the following, we discuss how to compress signatures and how to decompose signatures into smaller *partial signatures*.

4.2.2 Signature Compression

We focus on light-weighted node-level lossless compressions. There are other compression opportunities, such as lossy compression, inter-node compression and inter-cell compression, which will be discussed in Section 4.5.

Node Level Compression

To compress signatures, one can first rewrite the entire tree to a binary string, and then conduct compressions. Alternatively, one can first compress the bit array in each node, and then assemble them to a binary string. We refer the former as tree-level compression and the latter as node-level compression. In this thesis, we adopt the node-level compression for the following reasons: (1) There is a large room for compression at the node level. For example, with page size $4KB$, the value of M in R-tree node varies from 204 (for two dimensions) to 94 (for five dimensions) [36, 8]; (2) bit arrays in different nodes may have significantly different

characteristics, and one may achieve better compression ratio by adaptively choosing different compression scheme according to node properties; (3) node-level compression is good for efficient online computation since only the requested nodes need to decompressed; and (4) node-level compression can be easily integrated with signature decomposition (Section 4.2.3). For tree-level compression, conducting decomposition after compression is useless since each partial compressed piece cannot be recovered; while conducting compression after decomposition is also difficult since the post-compression size is not available for decomposition module.

Adaptive Coding

We apply three lossless compression methods: run-length encoding (**RL**) [32], position index encoding (**PI**) (instead of coding the original bit-array, **PI** encodes the positions of 1's.) and prefix compression encoding (**PC**) [31]. Since the bit-map compression problem is well studied, we skip the description of those methods. The details can be found in the references. To adaptively choose the best coding scheme for a node, we use a unified coding structure as shown in Figure 4.4. The first entry *CS* (coding scheme) indicates which compression scheme is used. The second entry *Len* indicates the length of the coding region. The last entry *coding region* stores the compressed bit array.

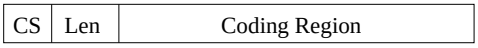


Figure 4.4: **A unified coding structure for a node**

In general, bit arrays among upper level nodes are dense (*i.e.*, contain many 1's) and those in lower level nodes are sparse (*i.e.*, contain many 0's). For each compression scheme, we implement two coding variations: *dense* and *sparse*. In the dense version, we encode 1's; and in the sparse version, we encode 0's.

As a result, we need 3 bits to code the *CS*: The first two bits represent three different compression methods (01, 10 and 11 for PI, RL, and PC, respectively) and the last one

indicates sparse (0) or dense (1) scenario. We use 000 for BL (*i.e.*, baseline coding). In the following, we examine the compression methods one by one. The example compressed results are shown in Table 4.2.

Scheme	CS	Len	Coding Region
BL	000	11011	01100000001100000000000001
RL	100	01110	01001101110011101110
PI	010	11001	00001000100101001011111011
PC	100	11001	00001011001001101111100011

Table 4.2: **Encoding a node with** $M = 32$

One Truncation: our first compression strategy is to remove the tailing 0's in the baseline coding. The example coding for sparse scenario is shown in Table 4.2. For dense bit array, we can remove the tailing 1s(*i.e.*, *One Truncation*). However, the straightforward removal may introduce ambiguity since we do not know how many 1's need be added back to recover the original bit array. For this purpose, we allocate the first $\lceil \log_2 M \rceil$ bits in the coding region to indicate the original length of the bit array. This coding strategy also applies on the rest compression schemes, and we do not state it explicitly thereafter.

Run-Length: This is a popular compression method for bitmaps. We represent a run (*i.e.*, a sequence of i 0's followed by a 1) by some binary encoding of the integer i. Since the binary representation of i needs $\lceil \log_2(i+1) \rceil$ bits, we first code $(\lceil \log_2(i+1) \rceil - 1)$ 1's and a single 0. Then, the value of i in binary follows. Using Table 4.2 as example, the first run is 01, where $i = 1$. The run-length code is 01: the first 0 indicates that i uses 1 bit, and the next 1 indicates that $i = 1$. In the dense version, a run means a sequence of i 1's followed by a 0. Since our signature generation method truncates tailing 0's in bit arrays, for correct recovery, we artificially add one 0 in the end of bit array for dense encoding. This strategy also applies on the rest compression schemes, and we do not state it explicitly thereafter.

Position Index: Here we compressed the baseline coding by only storing the positions of 1's in a sparse bit array. Each position takes $\lceil \log_2 M \rceil$ bits. In the dense version, we will store the position of 0's.

Prefix Compression: This is an improved version of the position index method. Let $n = \lceil \log_2 M \rceil$. The prefix compression groups all positions by their prefixes (*i.e.*, first p bits), and stores the suffixes (*i.e.*, rest $n - p$ bits) after each prefix. To get a binary coding, we first write p-bit prefix, which is followed by the number of suffixes under this prefix. Since there are at most 2^{n-p} suffixes, the number can be coded by $n - p$ bits (using one-less principle). Finally, we write individual suffixes, each of which takes $n - p$ bits. Given the value of n, the optimal value [31] of p is given by $\log_2 \frac{2^n}{n \ln 2}$. In Table 4.2, $n = 5$ and $p = 3$. 00001 and 00010 share a same prefix 000. The number of suffix is coded as 01 and the suffixes are 0110.

4.2.3 Signature Decomposition

In this subsection, we describe how to decompose the compressed signature into smaller *partial signatures* such that each partial signature fits in a data page with size P. In order to facilitate the incremental updates, we control the size of each partial signature around αP ($\alpha < 1$).

The decomposition procedure works as follows. Given a signature tree, we start from the root node and conduct the bread-first traversal. For each node, we apply all compression methods and pick the one giving best compression ratio. At the same time, we keep track of the accumulated size of traversed nodes. If the size reaches αP, we stop the traversal and the first partial signature is generated. Next, we start from the first child of the root, and conduct bread-first traversal within the subtree under this node. Nodes coded by previous partial signatures will be skipped. After finishing the first child, we continue on the following children of the root. If there are still nodes left after the second-level encoding, we will go to the third level, and so on. Each partial signature corresponds to a sub-tree, and is referenced by the SID (Section 4.2.1) of the root of that sub-tree.

We demonstrate the above algorithm using Figure 4.3.a as example. Starting from the root node, we generate the first partial signature which contains the root node (10) and the

second-level node (11). This partial signature is referenced by the root $SID(=0)$. We then start from the root's first child node N_1. The N_1 node has been coded and is skipped. The two leave nodes N_3 and N_4 are included. This partial signature is referenced by node N_1, whose $SID = 1$.

During query processing, we load the partial signatures p only if the node encoded within p is requested. To begin with, we load the first partial signature referenced by the root. When the query processing model requests a node n which is not presented in the current signature, we use the first level node in the path from the root to n as reference to load the next partial signature. If the partial signature has already been loaded, we check the second-level node in the path, and so on. For example, in Figure 4.3.a and Figure 4.1, suppose the a bit in the leaf node N_4 is requested but not presented in the current signature. The path from the root to N_4 is $\langle 1, 2\rangle$, and we load the partial signature referenced by $SID = 1$ (*i.e.*, node N_1).

Algorithm 1 The Cubing Algorithm

Input: A database D
A set of Boolean Dimensions: S
A set of ranking dimensions: R
A set of cuboids to be computed: C
Output: R-Tree and ranking-cube

1: Partition all data tuples using R-tree with R;
2: Generate *path*s for all tuples;
3: **for** each cuboid $c \in C$
4: Sort tuples according to dimensions in c;
5: **for** each $cell \in c$
6: Generate signature from tuple *path*s;
7: Compress and Decompose signature;
8: Store and index each partial signature by *cell* and the root SID of the partial signature;
9: **return**

4.2.4 The Cubing Algorithm

After presenting all key steps, we summarize the complete signature cubing procedure in Algorithm 1. Due to the curse of dimensionality, we may only compute a subset of low dimensional cuboids, as suggested by [47, 70]. Section 4.3.3 discusses how to assemble an arbitrary signature online.

Line 1 partitions data tuples by R-Tree, according to the ranking dimensions. Line 2 generates the *path* for each tuple (Section 4.2.1). For each cuboid c specified by the target cuboids set C, line 4 sorts the data according to dimensions in c. For each *cell* in the cuboid c (*i.e.*, a sub-list of tuples belonging to the *cell*), line 6 creates a tree-structured signature (Section 4.2.1). Line 7 traverses the signature tree, compresses the nodes (Section 4.2.2) and decomposes the signature into several partial signatures (Section 4.2.3). Line 8 stores and indexes the partial signatures by the attribute values in *cell* and the root SID's of the partial signatures.

4.2.5 Incremental Maintenance

Finally, we discuss incremental updates for signatures. We take insertion as an example since the processing for deletion and update is similar. Inserting a new tuple to R-tree may cause node splitting and tuple re-insertion [36, 8]. Before presenting the complete solution, we first discuss a simpler case where there is no node splitting nor tuple re-insertion.

Insertion without Node Splitting

Every node (including leaf) in R-tree can hold up to M entries. We assume each node keeps track of its free entries. When a new tuple is added, the first free entry is assigned. In case there is no node splitting nor tuple re-insertion, only the path of the newly inserted tuple is updated, and those for other tuples keep the same.

Suppose the database already has $t_1, t_2, t_3, t_5, \ldots, t_8$, a new tuple t_4 in inserted. Figure

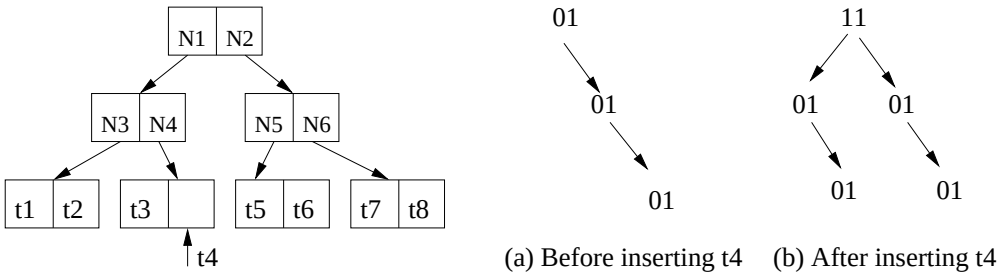


Figure 4.5: **Insertion without Node splitting**

4.5 demonstrates the change of $(A = a_3)$ signature (see Table 4.1). t_4 is first inserted into R-tree and finds an entry in leaf node N_4. A new $path = \langle 1, 2, 2 \rangle$ is computed for t_4. Since the *path*s for all other tuples did not change, only the signatures of cells a_3, b_3 and a_3b_3 in cuboids A, B, and AB are affected. Furthermore, within those signatures, only the entries on the path from the root to t_4 are possibly affected. We then load those partial signatures containing the *path*, and flip the corresponding entries from 0 to 1.

Insertion with Node Splitting

Here we discuss the case where the insertion causes node splitting or tuple re-insertion. When a node is split, the *path*s of all tuples under the split entry will change. To correctly update signatures, we need to collect the old and new *path*s for those tuples. We first traverse the sub-tree under the entry before splitting to get old *path*s, and then traverse it again after splitting to get new *path*s. Similarly, for tuples scheduled for re-insertion, we compute the old and new *path*s before and after re-insertion. As a result, there is a set of tuples whose *path*s are updated.

For example, suppose the database already has tuples as shown in Figure 4.6.a. The insertion of $t2$ causes the entry N_3 in the node N_1 being split. The *path* values of t_1, t_2 and t_3 may change, and they form an update set U. Before splitting, we get the old *path*s of t_1 and t_3 as $\langle 1, 1, 1 \rangle$ and $\langle 1, 1, 2 \rangle$, respectively. After splitting, their new *path*s are $\langle 1, 1, 1 \rangle$ and $\langle 1, 2, 1 \rangle$. The *path* for t_2 is $\langle 1, 1, 2 \rangle$. Since the *path* for t_1 does not change, t_1 is removed from U.

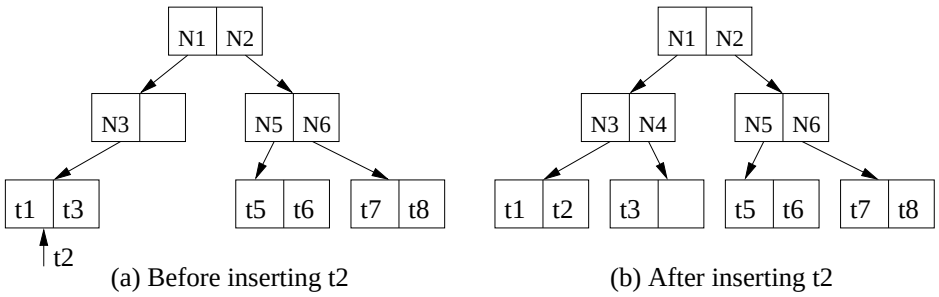


Figure 4.6: **Insertion with Node splitting**

Algorithm 2 describes the procedure for incremental updating signatures. Line 1 inserts tuples into database and R-tree, and returns the update set U, where each tuple has a new *path* and an old *path*. We identify the target cells (*i.e.*, signatures for updating) on lines 2 to 4. For each old *path*, line 6 clears the corresponding bit in the signature. Note we only clear the bit on the leaf node. If all the entries in a node n are 0, we then recursively clear the entry (in n's parent node) which points to n. For each new *path*, line 7 sets all bits along the path to 1. Finally, the updated signature is written back to the disk.

Algorithm 2 Incremental Maintenance

Input: The database D, R-tree T and ranking-cube C
A set of new tuples S to be inserted

1: Insert S to D and T, and get update set U;
2: **for** each cuboid $c \in C$
3: sort tuples in U according to dimensions in c
4: **for** each *cell* appearing in U;
5: load the signature for *cell*;
6: clear bits according to old *path*s ;
7: set bits to 1 according to new *path*s;
8: write back the updated signature;
9: **return**

4.3 OLAPing Ranked Queries

Having presented the *signature* measure and ranking-cube, we discuss how to use signatures in query processing.

4.3.1 Framework for Query Processing

Given the boolean and ranking criteria, an algorithm can first filter data tuples by boolean predicates and then compute the top-k (*i.e.*, boolean pruning first). Alternatively, one can search data tuples according to the ranking functions and verify the boolean constraints on each candidate (*i.e.*, ranking pruning first). Both approaches may retrieve data that will be pruned by the other criterion later, and thus are not efficient. To facilitate ranking pruning, we use a branch-and-bound search paradigm which starts from the root node of the R-tree and progressively expands a node by examining its child nodes. A node is pruned if its ranking score is lower than the current top-k results. The search halts when there is no node left.

To integrate boolean pruning in the above search framework, the algorithm needs to identify whether an underlying node contains any object satisfying the boolean predicates. This task consists of two challenges: (1) the boolean pruning needs to be conducted for *arbitrary* boolean predicates, and on *arbitrary* nodes from the root to leaves; and (2) the boolean pruning component should be fairly efficient and cannot be the bottleneck for the whole framework. The signature measure materialized in the last section can be used here.

4.3.2 Ranking Pruning

To begin with, we outline the *signature-based* query processing in Algorithm 3. The algorithm follows the branch-and-bound principle to progressively retrieve data nodes. To push ranking pruning deep into the database search, the nodes have to be scheduled in the best-first way. That is, nodes which are most promising for top scores need to be accessed

Algorithm 3 Framework for Query Processing

Input: R-tree R, ranking-cube C, user query Q

```
1: topk = φ; // heap with size k to store the result set
2: c_heap = {R.root}; // initialize candidate heap
3: while ((c_heap ≠ φ) and (f(topk.root) ≤ f(c_heap.root))
4:     remove top entry e;
5:     if (signature(e) == false), continue;
6:     if (e is a data object)
7:       insert e into topk;
8:     else // e is a node
9:       for each child e_i of e // expand the node
10:          insert e_i into c_heap;
11: return
```

first. Suppose the ranking function is f. We define $f(n) = \min_{x \in n} f(x)$, for each node n. Consequently, the candidate heap uses $f(n)$ to order nodes, and the root has the minimal score. The ranking pruning consists of two steps. First, for each candidate e (node or data object) submitted for prune checking, the ranking pruning will first compare $f(e)$ with all data objects in the top-k (line 3) list. If there are at least k objects whose scores are better than $f(e)$, then e can be pruned. Second, at any time, we can sort data objects in the top-k list according to their f scores, and only need to keep k results in the list (with the k^{th} result at the root of the top-k heap).

We briefly explain each step as follows. Line 1 initializes a heap (with size k) to store the final results, where the k^{th} data is at the root of the heap. Each node n is associated with a value $f(n)$, and the root of the *c_heap* contains an entry e with minimal (or best) $f(e)$. Line 5 checks whether e is pruned by the boolean predicate, for every e that passes the checking, e is a new top-k candidate if it is a data object (lines 6-7). Otherwise, the algorithm further examines e's child nodes (lines 9-10).

The correctness of the algorithm is ensured by two facts. First, if a node n ranks lower than k data objects, all child nodes of n are pruned. Second, for each node n, let $f(n) = \min_{x \in n}(f(x))$ be the lower bound value over the region covered by n. Clearly, a data object

t cannot be pruned by any data objects contained by node n if $f(t) \leq f(n)$. The results are generated on Line 7, and we shall show that top-k heap contains valid results. This is because: (1) e passes the boolean pruning, and thus e satisfies the boolean predicate; and (2) e passes the ranking pruning, and thus e is among the top-k results.

4.3.3 Boolean Pruning

The query processing has two main tasks: *ranking pruning* and *boolean pruning*. The boolean pruning is accomplished by signatures. No matter whether the entry e submitted to the *prune* procedure is a data tuple or an intermediate node, we can always get the *path* of e, and query the signature using the *path*. Since we only materialized a subset of cuboids, the ranking-cube may not have the signature ready for an arbitrary boolean predicate BP. To assemble a signature for BP online, we need two operators: *signature union* and *signature intersection*. Intuitively, given two signatures s_1 and s_2, the union operator computes the bit-or result and the intersection operator computes the bit-and result. Suppose the signature to be assembled is s. Any bit that is 1 in s_1 or s_2 will be 1 in s by the union operator. The intersection operator is defined in a recursive way. For each bit b in s, if either the corresponding bit in s_1 or s_2 is 0, the bit is 0. Otherwise, we will examine the intersection result of b's child nodes. If the intersection causes all bits in the child nodes being 0's, we will set b to 0. An example of signature assembling based on Table 4.1 is shown in Figure 4.7. Note in query execution, we only assemble the partial signatures that are requested by *prune* during query processing. To assemble signature for arbitrary BP, we assume that the ranking-cube always contains a set of *atomic* cuboids (*i.e.*, one-dimensional cuboids on each boolean dimensions).

Here we estimate the overall I/O cost of Algorithm 3 for ranked queries. The cost consists of two parts: C_{sig} and $C_{R\text{-}tree}$. The former represents the cost for loading signatures and the latter is the cost for loading R-tree blocks. In reality, one partial signature encodes many R-tree nodes. For example, a partial signature generally has size $4KB$ (*i.e.*, a page size), and

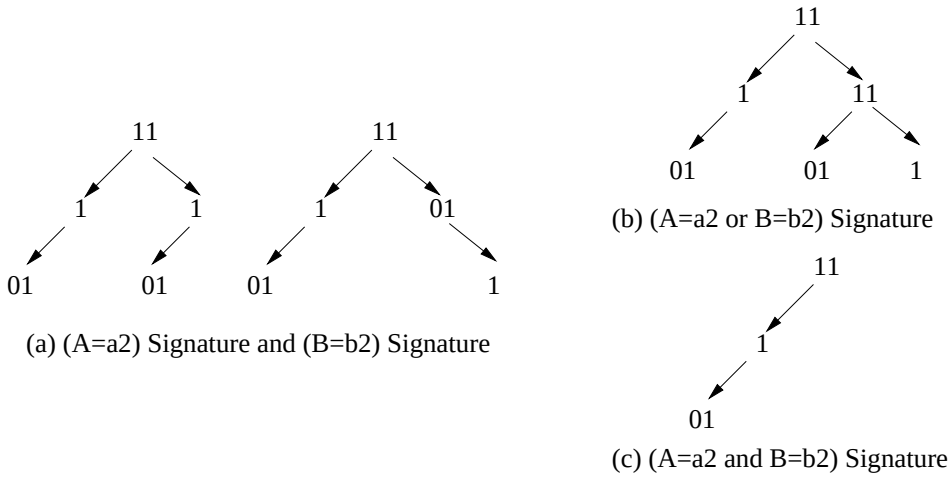


Figure 4.7: **Assembling signatures**

the size of each bit-array in the signature is up bounded by $\frac{M}{8}$ bytes (without compression). Here M is the maximal number of child entries in the node. For a 2 dimensional R-tree, $M = 204$ and a partial signature can encode 160 nodes. Thus, $C_{sig} \ll C_{R\text{-}tree}$, and we focus on $C_{R\text{-}tree}$ only. When $BP = \phi$ (*i.e.*, no boolean predicates), one can easily show that the progressive query framework demonstrated in Algorithm 3 is I/O optimal such that the algorithm only retrieves R-tree blocks that may contain top results. When $BP \neq \phi$, since the signature provides exact answers for boolean checking, Algorithm 3 only retrieves R-tree blocks that pass domination checking and boolean checking. We have the following claim.

Lemma 3 *For ranked queries with boolean predicates, Algorithm 3 is optimal in terms of* $C_{R\text{-}tree}$ *such that the number of R-tree blocks retrieved is minimized.*

4.4 Performance Study

This section reports our experimental results. We compare the query performance of Algorithm 3 with two other alternatives: (1) the *boolean-first* approach that evaluates boolean predicates before the preference criteria, and (2) the *ranking-first* approach that conducts boolean verification after each preferred candidate is generated. We first discuss the experimental settings, and then show the computation and space costs of ranking-cube, and online

query performance. This query processing framework will be used again later in this thesis to answer general preference queries, and we will show more details and experimental results in Chapter 7.

4.4.1 Experimental Setting

We defines the data sets and the experimental configurations for all approaches.

Data Sets

We use both synthetic and real data sets for the experiments. The real data set we consider is the *Forest CoverType* data set obtained from the UCI machine learning repository web-site (www.ics.uci.edu/∼mlearn). This data set contains $581,012$ data points with 54 attributes. We select 3 quantitative attributes (with cardinalities 1989, 5787, and 5827) as ranking dimensions, and other 12 attributes (with cardinalities 255, 207, 185, 67, 7, 2, 2, 2, 2, 2, 2, 2) as boolean dimensions. We also generate a number of synthetic data sets for our experiments. For each synthetic data, $\mathcal{D}_p$ denotes the number of ranking dimensions, $\mathcal{D}_b$ the number of boolean dimensions, $\mathcal{C}$ the cardinality of each boolean dimension, $\mathcal{T}$ the number of tuples, $\mathcal{S} = \{E, C, A\}$ the uniform, correlated and anti-correlated data distributions.

Experimental Configurations

We build all *atomic* cuboids (*i.e.*, all single dimensional cuboids on boolean dimensions) for ranking0cube. Signatures are compressed, decomposed and indexed (using B_+-tree) by cell values and SID's. The page size in R-tree is set as $4KB$. In the experiments, we compare our proposed signature-based approach (referred as *Signature*) against the boolean-first (referred as *Boolean*) and ranking-first (referred as *Ranking*) approaches. For simplicity, we discuss how *Boolean* and *Ranking* are implemented.

Boolean first: We use B_+-tree to index each boolean dimension. Given the boolean predicates in query, we first select tuples satisfying the boolean conditions. To process

boolean filters, we may use index scan or table scan, we report the best performance of the two alternatives. In the meanwhile, *Boolean* maintains a heap with size k to keep track the current top-k data objects. In this way, the memory requirement for *Boolean* is bounded by k.

Ranking first: The *Ranking* algorithm progressively retrieves R-tree blocks until the results are computed. The framework is similar to Algorithm 3, except that there is no boolean checking in the procedure. Since R-tree only keeps values in ranking dimensions, we build index on *tid* for the database and keep *tid* with each tuple in R-tree for boolean verification. The boolean verification involves random access and we only issue a boolean checking for a tuple in between lines 6 and 7 in Algorithm 3. That is, we only verify a tuple which has been determined as a candidate result. In this way, one can prove that the number of boolean verification is *minimized*. One may suggest to keep boolean dimensions in R-tree for cheap verification. This approach may not be a sound solution in reality: first, it reduces the capacity of each node, and the size of R-tree may increase a lot; and second, it may violate some constraints (*e.g.*, the tuples must be sorted by a primary key) and introduce difficulties for other types of data access (*e.g.*, sequential table scan).

4.4.2 Experimental Results

Experiments are conducted to examine (1) the construction and space costs for ranking-cube, (2) the performance on skyline queries, (3) the performance on top-k queries, and (4) the effect of boolean predicates on query performance, including the drill-down and roll-up queries.

Construction and Space Costs for ranking-cube

We first examine the cost to construct (Algorithm 1), store, and incrementally maintain (Algorithm 2) ranking-cube. We use synthetic data set in this set of experiments. By default, both the number of boolean dimension $\mathcal{D}_b$ and the number of ranking dimension $\mathcal{D}_p$

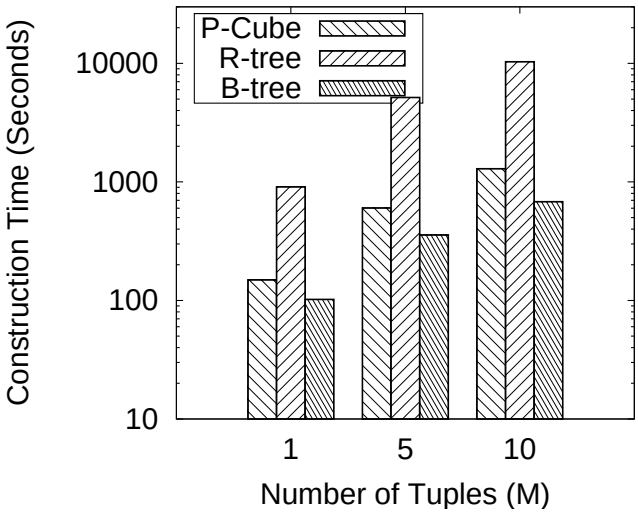


Figure 4.8: **Construction Time w.r.t.** $\mathcal{T}$

are 3. The cardinality $\mathcal{C}$ of each boolean dimension is 100, and the distribution $\mathcal{S}$ among ranking dimensions is uniform.

To study the scalability of Algorithm 1, we vary $\mathcal{T}$ (the number of tuples) from 1M to 10M. Figure 4.8 shows the costs to build R-tree partition, compute ranking-cube, and build B_+-tree index for all boolean dimensions. The R-tree is shared by both *Signature* and *Domination* approaches and B_+-trees are used by *Boolean* method. We observe that the computation of ranking-cube is 7-8 times faster than that of R-tree, and is comparable to that of B_+-tree. On the other hand, for space consumption, ranking-cube is 2 times less than B_+-trees and 8 times less than R-tree (Figure 4.9). Figure 4.10 further shows the effectiveness of node compressions (Section 4.2.2) by varying the cardinality $\mathcal{C}$ of each boolean dimension from 10 to 1000. The compressed ranking-cube is 3 times smaller than that uses baseline coding only. Note although the space cost for the entire ranking-cube increases with $\mathcal{C}$, the size of individual signature measure in each cell is significantly reduced.

To examine the performance of incremental update of ranking-cube, we run Algorithm 2 by inserting 1 to 100 new tuples. The execution times in Figure 4.11 show that the incremental maintenance algorithms are much better than the re-computation since we only

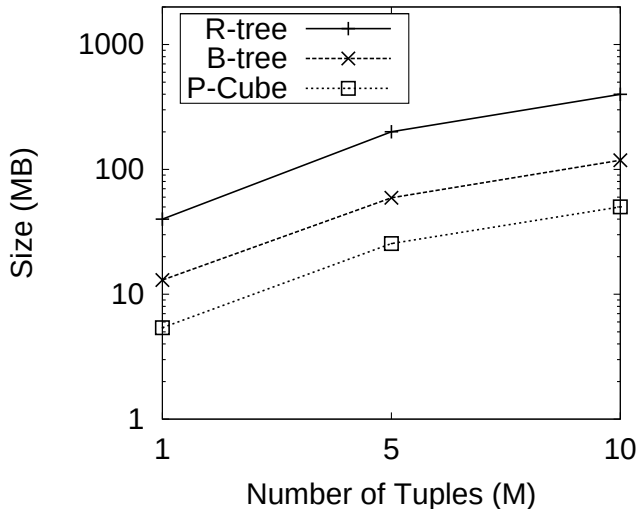


Figure 4.9: **Materialized Size w.r.t. $\mathcal{T}$**

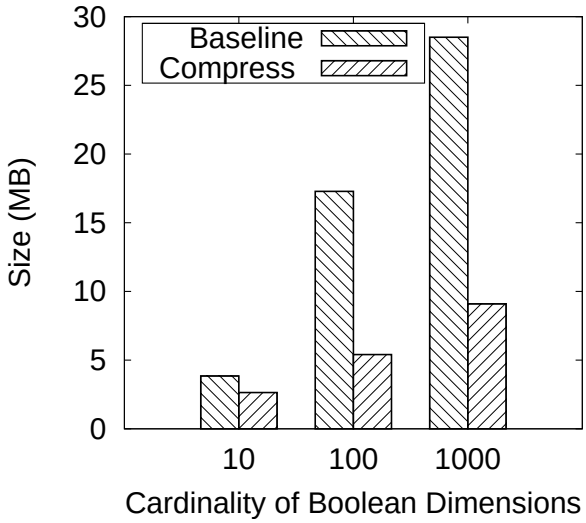


Figure 4.10: **Signature Compression w.r.t. $\mathcal{C}$**

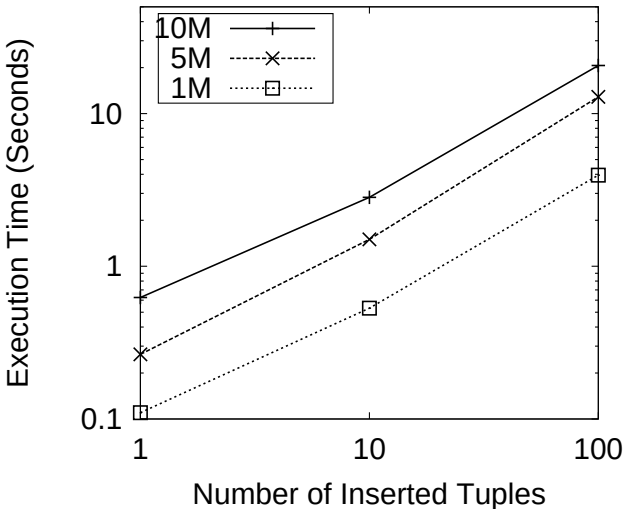


Figure 4.11: **Cost of Incremental Updates**

update target cells. Moreover, maintenance in batch is more scalable than maintenance tuple by tuple. For example, the execution time for inserting one tuple in $1M$ data is 0.11 Seconds. While the average cost for inserting 100 tuple on the same data is 0.04 Seconds.

Performance on Query Processing

After reporting the results on the off-line computation, we continue to evaluate the performance on top-k queries. We conduct two sets of experiments on query performance with respect to the value of k and the type of functions.

To demonstrate the query performance, we compare our signature-based approach with *Boolean* and *Ranking*. Suppose the ranking function is formulated on three attributes X, Y, and Z, and they are partitioned by an R-tree. For demonstration, we use three queries with controlled functions: (1) a *linear* query with function $f_l = aX + bY + cZ$, where a, b, and c are random parameters; (2) a *distance* query with function $f_d = (X - x)^2 + (Y - y)^2 + (Z - z)^2$ where x, y, and z are random parameters. This is a typical nearest neighbor query, which is frequently used in database systems; and (3) a *general* query with function $f_g = (2X - Y - Z)^2$. This query is often used to measure the min square error.

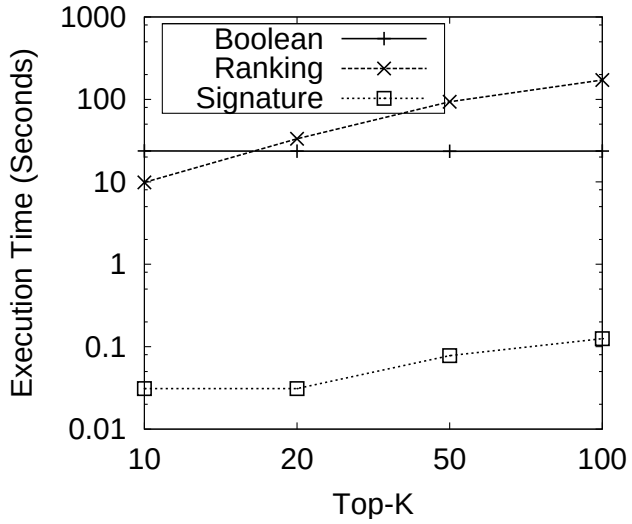


Figure 4.12: **Execution Time w.r.t.** k

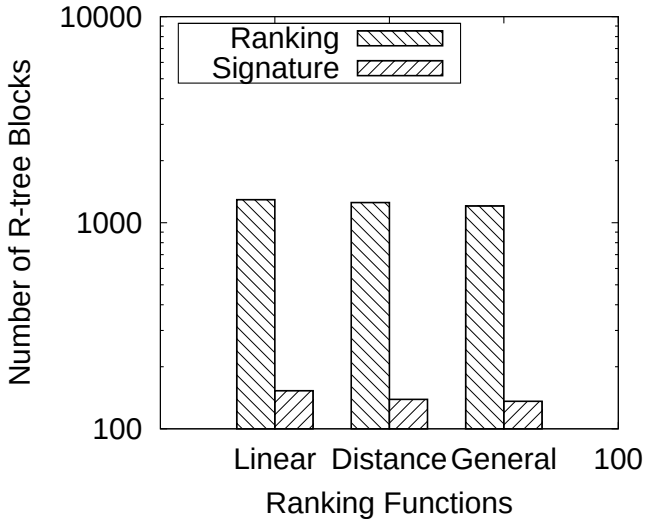


Figure 4.13: **Disk Access w.r.t. Functions**

We use $1M$ synthetic data and the query execution time on linear function with respect to different k values in Figure 4.12. We observe that *Boolean* is not sensitive to the value of k; *Ranking* performs better when k is small; and *Signature* runs order of magnitudes faster. We further demonstrate the number of R-tree block accesses by *Ranking* and *Signature* in Figure 4.13 (with $k = 100$), and the results show that *Signature* consistently improves the query performance with respect to all type of functions.

In summary, the signature based approach is the consist winner on both off-line and online computations. The ranking-cube only consumes small space and can be efficiently computed. In the meanwhile, it significantly speeds up the query performance since it takes advantage of both boolean pruning and domination pruning.

4.5 Discussion

Besides the lossless compression discussed in this chapter, lossy compression is also applicable. In this subsection, we discuss several lossy compression opportunities and the adaption of query processing.

A popular lossy compression methods is bloom filter [10], which uses k hash functions and maps an entry to k positions in a bit array. During the building phase, the corresponding bits are set to 1. At the query time, one can apply the k same hash functions to get k positions, and return true if and only if all of them are 1. False positives are possible. But it guarantees no false negative. To use bloom filter in our framework, we can build a bloom filter on all SID's whose corresponding entries are 1 in the signature. Signature decomposition is also possible such that we can build a bloom filter for each partial signature. During query execution, we can load the compressed signature (*i.e.*, a bloom filter), and test a SID upon that. If the bloom filter returns no, it is safe to prune the entry. Since it is possible that the filter fails to prune a tuple, we need the boolean verification step as that in *Domination*. Incremental maintenance is difficult because bloom filter only allows insertion,

but no deletion.

Other than bloom filter, we can compress the signature by exploiting the node similarities. For example, one can merge n consequent neighboring nodes to one single node by computing the bit-or result. To control the false positive rate, it is desirable to only select and merge similar nodes. To generalize, we can compress the ranking-cube by merging similar signatures. The union operator discussed in Section 4.3.3 can be used to compute the merged result, which will be referenced by both cells. Exploiting signature-level similarity is especially useful for data set which has strong correlation between boolean attributes.

Chapter 5

High Dimensional Data

In Chapter 3, we discussed the ranking fragment method to address the case of high boolean dimensions. In this chapter, we present our solution for the case of high ranking dimensions [69]. In database systems, the data partition is also referred to index. The methods demonstrated here is not confined to ranking cube framework only, and it can be applied to more general application scenarios in database systems. Consequently, we would like to present this chapter in a more general context by the topic of index-merge. Note in the special case of ranking-cube, the index-merge corresponds to online data partition assembling.

Our work is closed related to the family of *threshold algorithm* (*i.e.*, TA), which has been widely studied for efficiently computing top-k queries. TA uses a *sort-merge* framework that assumes data lists are pre-sorted, and the ranking functions are *monotone*. However, in many database applications, attribute values are indexed by tree-structured indices (*e.g.*, B-tree, R-tree), and the ranking functions are not necessarily monotone. To answer top-k queries with *ad-hoc* ranking functions, this chapter studies an *index-merge* paradigm that performs progressive search over the space of joint states composed by multiple index nodes.

We address two challenges for efficient query processing. First, to minimize the search complexity, we present a *double-heap* algorithm which supports not only progressive state search but also progressive state generation. Second, to avoid unnecessary disk access, we characterize a type of "empty-state" that does not contribute to the final results, and propose a new materialization model, *join-signature*, to prune empty-states. To demonstrate the significance of these two challenges, Table 5.1 shows a performance comparison between the basic index-merge and the improved algorithm developed in this chapter.The results are

collected on a top-100 query with ranking function $f = (A - B^2)^2$, which merges two B_+-tree indices on attributes A and B. The database has $1M$ tuples.

Index-Merge	States Generated	Disk Accesses
Basic	$420,323$	$4,133$
Improved	$9,237$	483

Table 5.1: **Significance of the two challenges**

The rest of this chapter is organized as follows. Section 5.1 presents the framework for query processing, and analyzes the problem. The double-heap algorithm and state expansion strategies are developed in Section 5.2, and the selective merge with join-signature is presented in Section 5.3. We report the experimental results in Section 5.4, discuss the extensions in Section 5.5.

tid	A	B	$f = (A - B)^2$
t1	10	40	900
t2	20	60	1600
t3	30	65	1225
t4	50	45	25
t5	54	10	1936
t6	72	30	1764
t7	75	36	1521
t8	85	62	529

Table 5.2: **A Sample Database**

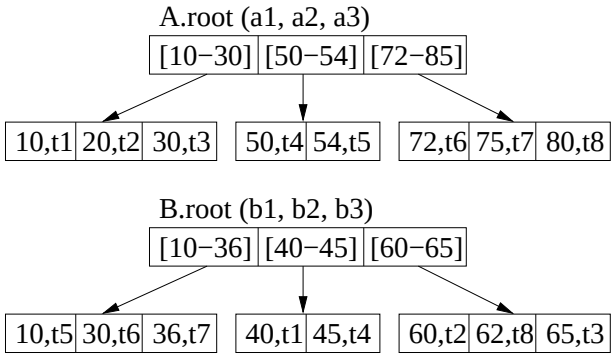
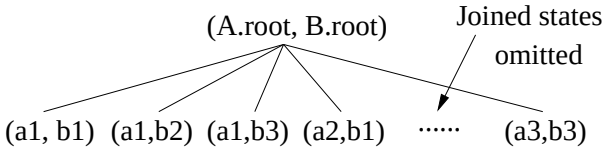


Figure 5.1: **Indices on A and B**



(a) Tree−structured joined states

State:	(A.root, B.root)	(a1, b1)	(a2,b2)	.....	(a3,b1)
f(S):	0	0	25	·····	1296

(b) States sorted by f(S)

Figure 5.2: **Space of Joint States**

5.1 Problem Analysis

5.1.1 Data Model

The task of finding top-k tuples from a database can be posed with either a maximization or a minimization criterion. Similar to previous chapters, we assume *minimization queries* are issued. Given a relation R with attributes $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$, a top-$k$ query with an evaluation function f, which is formulated on a subset of attributes $\{A'_1, A'_2, \ldots, A'_m\} \subseteq \mathcal{A}$, asks for k data tuples $t_1, t_2, \ldots, t_k$ such that for any other $t \in R$, $f(t) \geq \max_{i=1}^{k} f(t_i)$. Following our work in Chapter 4, we also assume that the ranking function f has the following property: *Given a function f and the domain region Ω on its variables, the lower bound of f over Ω can be derived.*

We assume attributes involved in f are indexed by some hierarchical indices (*e.g.*, B-tree or R-tree), such that a subspace occupied by a tree node is always contained in the subspace of its parent node. Suppose indices $\{I_1, I_2, \ldots, I_m\}$ are used to answer a top-k query. The space of joint states inherits the hierarchical property of the index. More specifically, the joint state can be recursively defined as follows. The root state is $(I_1.root, I_2.root, \ldots, I_m.root)$, and for each joint state $(I_1.n_1, I_2.n_2, \ldots, I_m.n_m)$, its child states are the Cartesian products of child nodes of $I_i.n_i$ $(i = 1, \ldots, m)$. If $I_i.n_i$ for some $i = 1, \ldots, m$ is a leaf node, $I_i.n_i$ itself is used in the Cartesian products to generate child states. If all $I_i.n_i$ are leaf nodes,

the joined state is a leaf state. Table 5.2 shows an example database, which consists of two attributes A and B (f is a demonstrating ranking function). Indices on A and B are shown in Figure 5.1 and the joint state space is assembled in Figure 5.2.a.

5.1.2 Framework for Query Processing

To compare with, we review the *sort-merge* algorithm used by TA. TA sequentially retrieves data from the sorted lists. At any state, we can classify tuples into three categories: fully merged, partially merged and unseen. The algorithm maintains an upper bound [1] for the current top-k fully merged tuples and a lower bound for the partially merged and unseen objects. If the upper bound score is no larger than the lower bound score, TA halts and returns the top-k results. Additionally, for those partially merged objects, unknown scores from some attributes can be looked up by random access, thus facilitating the early termination.

Similarly, in *index-merge*, we need to define two strategies: (1) a way to schedule node access, and (2) a stop condition that guarantees the top-k results are found. Given the value boundaries in index nodes, each joint state S is associated with a domain region $\Omega(S)$. We define $f(S)$ as the lower bound of the ranking function f over a joint space S. For example, in Figure 5.2, $\Omega(a1, b1) = [10, 30] \times [10, 36]$, and $f(S) = 0$ for ranking function $f = (A - B)^2$. Clearly, for each joint state, $f(S) \geq f(parent(S))$. The index-merge method starts with the joint root state, progressively finds the state with minimal $f(S)$ and examines its child states. In the meanwhile, the algorithm maintains an upper bound score of the current top-k results. The search stops when the upper bound score is no larger than the minimal $f(S)$ (for all rest S). The procedure is described in Algorithm 4.

We briefly explain the query processing as follows. The algorithm maintains two heaps. The $TopK$ heap keeps current best k results that are fully merged, and the root of the heap is the k^{th} tuple. The *g_heap* maintains candidate states for search, and the state with minimal

[1] Note we use minimization queries in this thesis.

Algorithm 4 Query Processing by Index-merge

Input: A set of Indices I, ranking function f, top-k

1: $TopK = \phi$; //heap with size k to hold current top-k
2: $g\_heap = \{Joint\ Root\}$; //heap for state search
3: **while** ($g\_heap \neq \phi$ *and* $f(TopK.root) > f(g\_heap.root)$)
4: remove top entry S from g_heap;
5: **if** S is a leaf state
6: Retrieve data and update $TopK$;
7: **else** //S is a non-leaf joint state
8: insert all child states of S to g_heap;
9: **return**

$f(S)$ appears at the root. We also maintain a hashtable h for tuple merge. Whenever a leaf state is retrieved, we look up tuples in h. If a tuple t is fully merged, and is better than $TopK.root$, we remove the root from $TopK$ and insert t to $TopK$.

5.1.3 Optimal Access Scheduling

We address two types of optimality for access scheduling: type-I that is CPU optimal and type-II that is disk-access optimal. By bridging Algorithm 4 and the optimal cases, we then identify two challenges.

Type-I Optimality

Suppose the upper bound score of final top-k results is s^* (remember that we use minimal k criteria in this thesis). *The type-I optimal scenario is that the algorithm only enumerates state S such that $f(S) \leq s^*$.* We denote the optimal number of states as $n_I^* = |\{S|f(S) \leq s^*\}|$. As an example, the top-1 query with function $f = (A-B)^2$ on the sample database (Table 5.2) returns t_4 as the final result. $s^* = 25$, and only $(A.root, B.root)$, $(a1, b1)$ and $(a2, b2)$ need to be enumerated. The states generated by Algorithm 4 can be classified into: (1) *examined states* that appear on Line 4; and (2) *generated candidates* that appear on Line 8. We have the following lemma.

Lemma 4 For a top-k query involving m indices, the number of examined states by Algorithm 4 is n_I^*, and the number of generated candidates by Algorithm 4 is upper bounded by $n_I^* \prod_{i=1}^{m} M_i$, where n_I^* is the optimal number of states given by type-I optimality, and M_i is the node fanout of index i.

proof We prove the claim for examined states, and the claim for generated candidates naturally follows. We show that for each examined state S, $f(S) \leq s^*$. At any stage, if $f(TopK.root) = s^*$, $f(S) \leq s^*$ is ensured by the while condition on Line 3. If $f(TopK.root) > s^*$, the top-k results have not been completely retrieved. Since S appears at the root of *g_heap* and $f(S)$ is the lower bound value of all future tuples, we have $f(S) \leq s^*$. ■

While the number of examined states is type-I optimal, the number of generated candidates is significantly higher. Fixing the page size as $4kB$, the fanout of B-tree node is 204 [32]. The number of child states (*i.e.*, Cartesian product) of two B-tree indices is up to 4.2×10^4, and that of three B-tree indices is up to 8.5×10^6. This is clearly non-trivial cost in terms of both CPU and memory. To avoid full expansion, an alternative method that uses unidirectional expansion was discussed in [39]. To expand a joined state $S = (a, b)$, either a is paired with all child nodes of b or b is paired with all child nodes of a. In this way, the number of child states is limited by the fanout of the node. However, the states generated by unidirectional expansion are less precise than those generated by full expansion, and thus the access scheduling may not be optimal. We identify the first challenges: *how to efficiently generate candidate states while preserving the optimal (type-I) access scheduling?*.

Type-II Optimality

To achieve type-II optimality, we first characterize two types of states: *redundant state* and *empty state*. An index node may appear in many joint states, and thus may be requested for retrieval for multiple times. We say a leaf index node is redundant if it has been retrieved previously. A non-leaf index node is redundant if all child nodes are redundant. Conse-

quently, a joint state is redundant if all composing index nodes are redundant. Redundant states need not to be retrieved because all data tuples contained by them have already been seen by the query processing algorithm. Since many index implementations buffer the previously retrieved index nodes, the redundant nodes (and thus the redundant states) can be identified on the fly. If the nodes are not buffered, the algorithm can maintain the IDs of those redundant index nodes in a hash-table.

To illustrate the definition of empty state, we build a mapping between data tuples and joint leaf states. We say a leaf state *contains* a tuple t if t appears in all the leaf index nodes joining the leaf state. A tuple may partially appear in many leaf states. However, there is a unique leaf state that contains it. On the other hand, a leaf state which contains one or more tuples is called *non-empty state*. Otherwise, it is an *empty state*. For example, in Figure 5.2.a, $(a1, b1)$ is an empty state and $(a2, b2)$ is a non-empty state. The definition of empty (non-empty) state can be recursively extended to non-leaf state as follows: A state is empty if all child states are empty; otherwise, it is non-empty. Let $S(t)$ be the leaf state containing tuple t. The following lemma shows a necessary condition of access scheduling using index-merge framework.

Lemma 5 For any top-k query that is processed by the index-merge framework, suppose the final results are $t_1, t_2 \ldots, t_k$, and $f(S(t_i)) < f(t_i)$ for all $i = 1, \ldots, k$. The leaf states $S(t_1), S(t_2), \ldots, S(t_k)$ must be retrieved when the query execution terminates[2].

proof In Algorithm 4, the query processing terminates when $\max_{i=1}^{k} f(t_i) \leq f(g\_heap.root)$, for current top-$k$ results $t_1, \ldots, t_k$. Assume $S(t_i)$ has not retrieved. We have $f(g\_heap.root) \leq f(S(t_i))$, which contradicts with $f(S(t_i)) < f(t_i) \leq \max_{i=1}^{k} f(t_i) \leq f(g\_heap.root)$. ∎

Since each t_i in the top-k results will be retrieved from non-empty state $S(t_i)$, it is safe to prune those empty-states. Thus, *the type-II optimal scenario is that an algorithm only retrieves states S such that: (1) $f(S) \leq s^*$, (2) S is not empty, and (3) S is not redundant*. We refer to

[2] $f(S(t_i)) = f(t_i)$ is possible for some special case. Here we simplify the analysis by assuming $f(S(t_i)) < f(t_i)$.

n^*_{II} as the optimal number of states given by type-II optimality. Following our example in Figure 5.2.b, we may skip state $(a1, b1)$, and only retrieve state $(A.root, B.root)$ and $(a2, b2)$.

We demonstrate the importance of pruning empty states by examining the leaf-states only. Suppose the database has $1M$ tuples, and each B_+-tree index contains at least $5,140$ leaf nodes (with fanout 204). Merging two indices leads to 2.64×10^7 leaf-states in total. Among them, there are at most $1M$ non-empty states. Thus, the probability that a leaf-state is empty is more than 96%.

The sort-merge framework also has empty-state problem, where objects near top on one list may rank low in other lists. To avoid accessing the large portion of data which is in the middle of the sorted lists, random access is used to directly jump to the bottom of the lists and resolve missing values. In the following, we discuss why random access is not applicable in the index-merge framework. As presented in the last subsection, tuples can be classified into three categories: fully merged, partially merged and unseen. Let s_k be the upper bound score for current top-k fully merged tuples, s_p and s_u be the lower bound scores for partially merged and unseen tuples. The search terminates when $s_k \leq \min(s_p, s_u)$. In sort-merge, we have $s_p \leq s_u$ because of the monotonicity of the ranking function. By issuing random accesses on partially merged tuples, s_k may decrease and s_p may increase. As the result, the termination condition may be satisfied without continuing to sequentially scan lists. While in index-merge, $s_u = s_p = f(g\_heap.root)$ because for each partially merged and unseen tuple t, $S(t)$ has not been retrieved. Issuing random accesses on partially merged tuples may decrease s_k to s'_k. However, since for all partially merged t, $f(t) \geq f(S(t)) \geq f(g\_heap.root)$, we have $s'_k \geq f(g\_heap.root) = s_u$. We conclude that random access in index-merge does not lead to early termination.

Different from redundant states, which can be checked on the fly, the empty state can only be identified with the assistance of some pre-computed module. Moreover, this module needs to be light-weighted. We identify the second challenge: *how to effectively prune empty-states with low overhead?*

5.2 Progressive Merge

This section presents our solution for the first challenge. In Algorithm 4, to search for the best child state, a parent state is fully expanded (on Line 8). In fact, most of them are never examined (on Line 4). For example, in Figure 5.2.a, there are 9 child states expanded by $(A.root, B.root)$, and only $(a1, b1)$ and $(a2, b2)$ are examined. In fact, to ensure the type-I optimality, we only need to compute the *next best* child state, for a given state S. We abstract[3] the interface that fulfills this requirement as $S.get\_next$.

In the rest of this section, we first discuss a double-heap method that integrates $S.get\_next$ with Algorithm 4, and then discuss two implementations for $S.get\_next$.

5.2.1 The Double-heap Algorithm

Let $S.num$ indicate how many times the $S.get\_next$ is called. In our example, suppose $S = (A.root, B.root)$. The first call of $S.get\_next$ returns $(a1, b1)$ and the second call of $S.get\_next$ returns $(a2, b2)$. $S.get\_next$ returns nothing when all child states are returned. For simplicity, we denote S_{num} as the num^{th} best child state of S according to $f(S_{num})$.

Before we proceed to the implementation of $S.get\_next$, we first discuss how to integrate $S.get\_next$ to Algorithm 4. To distinguish the integrated algorithm from Algorithm 4, we refer to the new algorithm as *dheap* (*i.e.*, double-heap). Instead of fully expanding S, and then discarding S (on Line 8 of Algorithm 4), *dheap* gets S_{num} (by calling $S.get\_next$) and inserts both S_{num} and S to *g_heap*. We keep S in *g_heap* because the query execution may further request the next best child state of S. Consequently, to reflect the status of partial expansion, we update $f(S) = f(S_{num+1})$, which is the best possible score for all future child states. Following our example in Figure 5.2, when $S = (A.root, B.root)$ needs to be expanded, we get $S_1 = (a1, b1)$, update $f(S) = f(S_2) = f(a2, b2)$ and insert both S_1 and S into *g_heap*.

[3]Here we use the object-oriented programming convention.

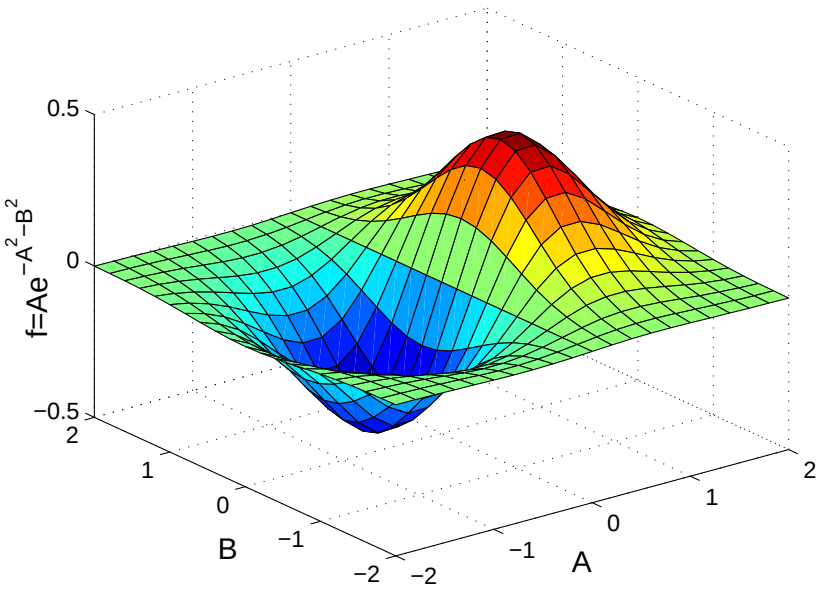


Figure 5.3: **Local Monotonicity**

The complete framework for progressive and selective index merge is outlined in Algorithm 5, where Lines 1 to 15 are the updated query processing procedure, and Lines 21 to 34 are the *S.get_next* method. Particularly, lines 5-6, 22-24 and 32-33 exploit pruning empty-states by join-signatures. The code is put here for the completeness of the algorithm, and the details will be discussed in the next section. The key components are *neighborhood_expand* and *threshold_expand*. The former works for some special scenario such that the best child state can be analytically found; and the latter is applicable on general cases. To avoid generating the same states along the multiple calls of *S.get_next*, each S maintains the status of child states that have been already generated, using a heap (*i.e.*, *l_heap* in *S.get_next*). We refer to this heap as *local heap*, in contrast to the *global heap* (*i.e.*, *g_heap* on Line 2) used in the main query processing loop. In the following subsections, we describe the two progressive search strategies one by one. For simplicity, we demonstrate both methods by merging two B-tree indices. The generalization to multiple indices is straightforward.

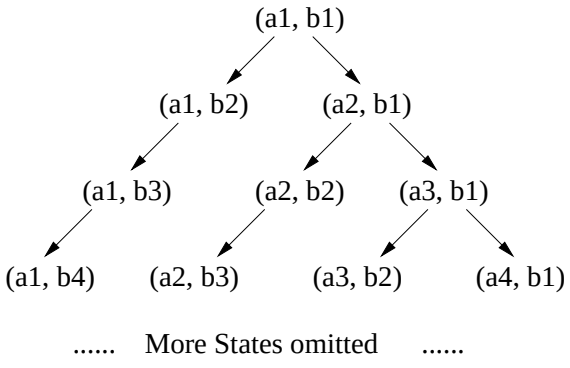


Figure 5.4: **Neighborhood Expansion**

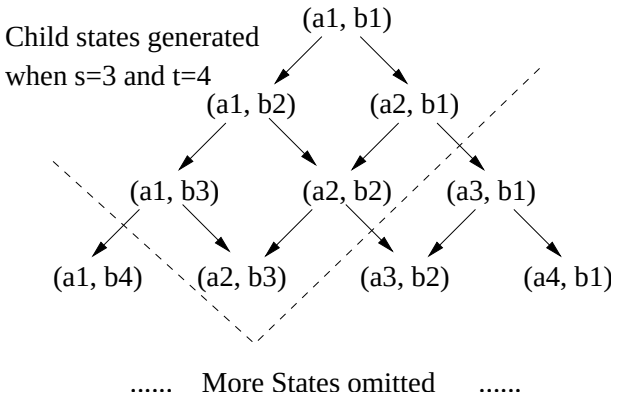


Figure 5.5: **Threshold Expansion**

5.2.2 Neighborhood Expansion

Although an ad-hoc ranking function does not perform regularly over the entire region, it may have monotonicity or semi-monotonicity in some sub-regions. We say a function f is *semi-monotone* if $f(x_1, x_2, \ldots, x_m) \leq f(x'_1, x'_2, \ldots, x'_m)$ whenever $|x_i - o_i| \leq |x'_i - o_i|$, for every i. f achieves minimal value at $(o_1, o_2, \ldots, o_m)$. For example, Figure 5.3 shows the plot of function $f = Ae^{-A^2-B^2}$ over two attributes A and B. The function is neither monotone nor semi-monotone over the entire domain. However, in sub-region $(A = [-2, 0], B = [-1, 1])$, f is semi-monotone; and in sub-region $(A = [1, 2], B = [-2, -1])$, f is monotone. The index-merge method recursively partitions the joint space and it is very likely that an ad-hoc function f becomes monotone or semi-monotone within a joint state. We present the neighborhood expansion for these special cases.

Since the neighborhood expansion requires a total order to be defined on the entries of an index node, the method may not be used on R-tree indices. Let $S = (A_1, B_1)$ be the state to be expanded. $a_1, \ldots, a_n$ and $b_1, \ldots, b_l$ are child entries of index node A_1 and B_1, respectively. Without loss of generality, we assume both a_i $(i = 1, \ldots, n)$ and b_i $(i = 1, \ldots, l)$ are sorted by attribute values. Suppose f is monotone over S and it achieves minimal value on (a_1, b_1) (*i.e.*, initial state, referred as $I(S)$). Thus, we have $f(a_i, b_j) \leq \min(f(a_{i+1}, b_j), f(a_i, b_{j+1}))$. The neighborhood expansion starts with the initial state and progressively generates neighboring states. A straightforward definition for the neighboring states of (a_i, b_j) is to enclose both (a_{i+1}, b_j) and (a_i, b_{j+1}). Since a state (a_{i+1}, b_{i+1}) can be generated by either (a_i, b_{i+1}) or (a_{i+1}, b_i), this approach requires duplicate-checking, which incurs additional overhead. Alternatively, we can define the neighborhood of a child state (a_i, b_j) as:

$$N(a_i, b_j) = \begin{cases} \{(a_i, b_{j+1})\} & if \quad 1 < j < l \\ \{(a_{i+1}, b_j), (a_i, b_{j+1})\} & if \quad j = 1, i < n \\ \phi & otherwise \end{cases}$$

The definition of N is illustrated in Figure 5.4. Clearly, there is no duplicate states to be generated. Whenever (a_i, b_j) appears at the root of the local heap and is to be returned by $S.get\_next$, we will insert $N(a_i, b_j)$ into the local heap. The procedure of neighborhood expansion is displayed as *neighborhood_expand* in Algorithm 6.

We briefly discuss how to extend the method to semi-monotone f. Suppose the extreme point of f is $o^* = (x^*, y^*)$. If o^* falls in a child state (a_s, b_t), the initial state is $I(S) = \{(a_s, b_t)\}$. Otherwise, we can find an i (j) such that a_i and a_{i+1} (b_j and b_{j+1}) enclose x^* (y^*). Accordingly, $I(S) = \{(a_i, b_j), (a_{i+1}, b_j), (a_i, b_{j+1}), (a_{i+1}, b_{j+1})\}$. We define the neighborhood for $I(S) = \{(a_s, b_t)\}$ case as follows, the definition for the other case is similar:

$$N(a_i, b_j) = \begin{cases} \{(a_i, b_{j+1})\} & if\ t < j < l \\ \{(a_i, b_{j-1})\} & if\ 0 < j < t \\ \{(a_{i+1}, b_j), (a_i, b_{j+1}), & \\ (a_i, b_{j-1})\} & if\ j = t, s < i < n \\ \{(a_{i-1}, b_j), (a_i, b_{j+1}), & \\ (a_i, b_{j-1})\} & if\ j = t, 0 < i < s \\ \{(a_{i-1}, b_j), (a_i, b_{j+1}), & \\ (a_{i+1}, b_j), (a_i, b_{j-1})\} & if\ i = s, j = t \\ \phi & otherwise \end{cases}$$

In general, to merge m indices, the cardinality of neighborhood for monotone functions is up to m and that for semi-monotone functions is up to $2m$. Lemma 6 gives the computational performance of the neighborhood expansion.

Lemma 6 Suppose f is monotone (or semi-monotone) over the entire domain, and there are m indices to be merged. The number of states generated by neighborhood expansion is upper bounded by mn_I^* for monotone functions and $2mn_I^*$ for semi-monotone functions, where n_I^* is the type-I optimal number of states to be generated (Section 5.1.3).

proof According to Lemma 4, $S.get\_next$ will be called at most n_I^* times in Algorithm 5. Each time when the $S.get\_next$ is called, the neighborhood expansion will generate up to m child states for monotone f and up to $2m$ child states for semi-monotone f. The conclusion follows. ∎

5.2.3 Threshold Expansion

Here we discuss the more general threshold expansion. Different from the neighborhood expansion, where the initial and consequent child states can be analytically located, threshold expansion conducts searching over the child state space. Suppose the state to be expanded is (A_1, B_1), and their entries are $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_l$, respectively. We define $f'(a_i) = f(a_i, B_1)$ $(i = 1, \ldots, n)$, which is the best score that a_i can achieve by pairing with any b_j $(j = 1, \ldots, l)$. Similarly, we define $f'(b_j) = f(A_1, b_j)$ $(j = 1, \ldots, l)$. We sort a_i and b_j in ascending order of $f'(a_i)$ and $f'(b_j)$ values, respectively. Without loss of generality, we assume the sorted orders are $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_l$.

The threshold expansion uses a sort-merge paradigm. We start search by inserting (a_1, b_1) to *l_heap* (*i.e.*, local heap), and use two variables s and t to keep track of the next positions on a_i and b_j sorted lists (*i.e.*, threshold position). Initially, $s = t = 2$. The stop condition is $f(l\_heap.root) \leq \min(f'(a_s), f'(b_t))$. At that time, the *l_heap.root* contains the next best child state. While the stop condition does not hold, the algorithm conducts progressive search. Suppose $f'(a_s) < f'(b_t)$, the algorithm creates $t - 1$ new child states (a_s, b_j) $(j = 1, \ldots, t-1)$, inserts them to *l_heap*, and then increases s by 1. For example, when $s = 3$ and $t = 4$, the states that have been generated are shown in Figure 5.5.

In general, suppose S is joined by m index nodes $(n_1, \ldots, n_m)$, and each n_i consists of child entries $e_i^1, \ldots, e_i^{M_i}$ (M_i is the fanout of node n_i). We define $f'(e_i^k) = f(n_1, \ldots, e_i^k, \ldots, n_m)$. The algorithm maintains a threshold position t_i for each n_i. Whenever a t_s is selected to advance, we generate the new child states by the Cartesian product $[e_1^1, \ldots, e_1^{t_1-1}] \times \ldots [e_s^{t_s}] \times$

$\ldots [e_j^1, \ldots, e_m^{t_m-1}]$. The algorithm is shown in Algorithm 6 as *threshold_expand*.

A function f is *general* over the state S if for any t_i $(i = 1, \ldots, m)$, the lower bound of $f(e_1^{t_1}, \ldots, e_m^{t_m})$ can only be derived by $\min_{i=1}^{m}(f'(e_i^{t_i}))$. Accordingly, we refer to algorithms performing child state search under this assumption as *general algorithms*. The following lemma shows that the threshold expansion is instance optimal [30] with optimal ratio 2^m, where m is the number of indices to be merged.

Lemma 7 Assume f is general over state $S = (n_1, \ldots, n_m)$, and for each node n_i, its entries satisfy $f'(e_i^1) < f'(e_i^2) < \ldots < f'(e_i^{M_i})$. The *threshold_expand* (or *te*) is instance optimal such that $n_{te} \leq 2^m n_{alg}$, where n_{te} is the number of child states generated by *te*, n_{alg} is the number of child states generated by any other general algorithm *alg* that correctly finds the next best child state.

proof Suppose *te* halts at t_i on each node n_i, and the next best child state is $nb = (e_1^{s_i}, \ldots, e_m^{s_m})$. Clearly, $s_i < t_i$ for all $i = 1, \ldots, m$. We show that for all i, $f'(e_i^{t_i-2}) < f'(e_i^{t_i-1}) \leq f(nb)$. Assume the threshold position at node n_k $(k \neq i)$ is t'_k when *te* decides to advance to $t_i - 1$ on node n_i. If $f'(e_i^{t_i-1}) > f(nb)$, we have $s_i < t_i - 1$ and there must exist one k such that $s_k \geq t'_k$. Otherwise, the best state nb has already been generated by *te* and $t_i - 1$ will not be selected. On the other hand, we have $f'(e_i^{t_i-1}) \leq f'(e_k^{t'_k}) \leq f'(e_k^{s_k}) \leq f(nb)$, which leads to contradiction. Since f is *general* over S, any general algorithm *alg* that correctly finds the best state has to generate and check all states from $[e_1^1, \ldots, e_1^{t_1-2}] \times \ldots \times [e_m^1, \ldots, e_m^{t_m-2}]$. Thus, $n_{te} = \prod_{i=1}^{m}(t_i - 1) \leq 2^m \prod_{i=1}^{m}(t_i - 2) = 2^m n_{alg}$. ∎

5.3 Selective Merge

In this section, we address the challenge towards the type-II optimality. To prune empty-states in merging indices $I_1, I_2, \ldots, I_m$, we materialize a *join-signature* of these m indices. Suppose the database has J indices. An ideal scenario is to compute a join-signature on any combination of two or more indices, and this leads to $(2^J - J - 1)$ different join-signatures

in total. In reality, one can only compute join-signatures on index combinations which are possibly to be queried. Alternatively, one can also materialize join-signatures on each pair of indices only, and use them to answer arbitrary queries. In the rest of this section, we discuss what is join-signature, how to compute join-signature and how to use join-signature during query processing.

5.3.1 Join-Signature

The join-signature is composed by the *state-signatures* of all non-leaf and non-empty states. For each non-leaf and non-empty state $S = (n_1, \ldots, n_m)$, we define $card(S) = \prod_{i=1}^{m} M_i$ as the cardinality of the child states, where M_i is the fanout of node n_i. The state-signature is an m-way bit array, where each entry corresponds to a child state. If a child state is not empty, the entry is set to 1; otherwise, it is 0. Using our example in Figure 5.2, only $S = (A.root, B.root)$ is a non-leaf and non-empty state. The state-signature of S is a 3×3 bit array, as shown in Figure 5.6.

	b1	b2	b3
a1	0	1	1
a2	1	1	0
a3	1	0	1

Figure 5.6: **signature**

tid	*A.path*	*B.path*
t1	$\langle 1 \rangle$	$\langle 2 \rangle$
t2	$\langle 1 \rangle$	$\langle 3 \rangle$
t3	$\langle 1 \rangle$	$\langle 3 \rangle$
t5	$\langle 2 \rangle$	$\langle 1 \rangle$
t4	$\langle 2 \rangle$	$\langle 2 \rangle$
t6	$\langle 3 \rangle$	$\langle 1 \rangle$
t7	$\langle 3 \rangle$	$\langle 1 \rangle$
t8	$\langle 3 \rangle$	$\langle 3 \rangle$

Table 5.3: **Pathes on Indices**

The space consumption of the join-signature depends on the number of non-leaf and non-empty states, as well as the size of each state-signature. We discuss these two issues one-by-one. As defined in Section 5.1.3, a leaf-state is non-empty if it contains at least one tuple. On the other hand, since tuples are exclusively distributed into different leaf-nodes in an index, tuples are also exclusively contained by different leaf-states. Suppose the database has T tuples. We conclude that there are at most T non-empty leaf-states, and at most $(d-1)T$ non-empty and non-leaf states (where d is the maximum depth among all joined indices).

We then discuss how to control the size of each state-signature, such that each of which can be stored within size P. Typically, P can be set as the page size. Given a state S, the value of $card(S)$ (*i.e.*, the number of child states) varies significantly from merging two indices to merging three indices. If $card(S) \leq P$, we store the state-signature as it is, and possibly, put neighboring state-signatures together if their accumulative size does not exceed P. Since state-signatures are essentially bit-maps, one can further apply bit-map compression methods, such as run-length encoding [32] and prefix-compression encoding [31], to reduce the sizes. We omit the details in this chapter.

When $card(S) > P$, we use bloom filter [10] for compression. Bloom filter uses k hash functions and maps an entry to k positions in a bit array. During the building phase, the bits at those k positions are set to 1. At the query time, one can apply the k same hash functions to get k positions, and return true if and only if all of them are 1. False positives are possible. But it guarantees no false negative. To use bloom filter for state-signature, we set the array size as $b(\leq P)$, and insert to the bloom filter all "1" entries in the state-signature. During query execution, if the bloom filter returns false for a child state c, c must be an empty-state. Suppose the number of non-empty child nodes (*i.e.*, "1" entries) is ne. The optimal choice [10] for k is $\frac{b}{ne} \ln 2$. To control the computational complexity (*i.e.*, the number of hash functions), we set the maximum number of hash function as $\bar{k}$. Consequently, $b = \min(P, \frac{\bar{k} \times ne}{\ln 2})$.

To reference a state-signature, we compute a unique key for each state $S = (n_1, n_2, \ldots, n_m)$ as follows. A level-k node n is associated with a path from the root (level 1) to n's parent (level $k-1$). The path consists of a sequence of entry positions: $path(n) = \langle p_1, p_2, \ldots, p_{k-1} \rangle$, where p_i is the entry position at the level-i node. For example, node a_1 in index A (Figure 5.1) corresponds to $path(a_1) = \langle 1 \rangle$. The key of S is a combination of paths of n_i: $key(S) = (path(n_1), \ldots, path(n_m))$. In implementation, one can one-to-one map $key(S)$ to a string or an integer. For each join-signature, we build an index on $key(S)$ for state-signatures.

5.3.2 Computing Join-Signatures

A naïve method to compute the join-signature is to traverse the joint state space and find the non-empty and non-leaf states. This is obviously not a scalable solution, since the space of joint states grows exponentially with the number of indices. Here we present a tuple-oriented approach.

Similar to the path generation for nodes, we can also compute a path for each tuple t. Given an index with depth d, the path of a tuple t is $path(t) = \langle p_1, \ldots, p_d \rangle$. Since the joined space is defined on the node granularity, we only need to know which leaf-node contains t. Hence, we can ignore the position on the leaf node, and $path(t) = \langle p_1, \ldots, p_{d-1} \rangle$. The paths for tuples in the sample database (Table 5.2) are shown in Table 5.3.

Treating each index as a dimension, and tuple paths as values, we compute the join-signature by recursive-sorting. Suppose we need to compute the join-signature for m indices $(I_1, \ldots, I_m)$, and the depth of each index is d_i $(i = 1, \ldots, m)$. According to the path generation discussed above, each tuple path has $(d_i - 1)$ entries in dimension I_i. To compute the state-signature for joint root state, we sort the tuples according to the first path entry on each I_i (*i.e.*, first compare $I_1.p_1$, then $I_2.p_1$, and so on). As an example, Table 5.3 is sorted by the first path entries. We then scan the sorted tuple list again, and insert distinct $(I_1.p_1, I_2.p_1, \ldots, I_m.p_1)$ into the state-signature, which is implemented by a bit array or a bloom filter. For each sub-list of tuples that share the same $(I_1.p_1, I_2.p_1, \ldots, I_m.p_1)$, we

recursively sort it on the second path entry on I_i. At the same time, $(I_1.p_1, I_2.p_1, \ldots, I_m.p_1)$ is the key to reference the next state-signature.

The above method is similar to some sorting-based data cube computation methods [9]. In fact, computing multiple join-signatures is indeed a multi-dimensional aggregation problem. As we mentioned earlier, suppose the database has J indices and we may compute up to $(2^J - J - 1)$ different join-signatures. Some techniques for efficient data cubing can be directly applied in our problem. First, when database is too large to fit in memory, one can partition the database (*e.g.*, by $I_1.p_1$) and compute each partial database [48]. Secondly, we can share the sorting on common indices [9] when computing multiple join-signatures. Finally, when J is large and there are too many join-signatures, we can only compute the low-dimensional (*e.g.*, pairwise) join-signatures [70]. The low-dimensional join-signatures can partially fulfill the task to prune empty-states in high-dimensional index-merge, and this will be addressed in the next subsection.

5.3.3 Pruning Empty-State by Join-Signature

After presenting the join-signature and its computation, we now discuss how to use it to prune empty-states during query processing. Candidate states are generated by the *S.get_next* procedure. To avoid generating empty-states, join-signatures can be integrated into *S.get_next* procedure as shown in Algorithm 5. When *S.get_next* is called for the first time, we load the state-signature for S using $key(S)$ (Line 22). For each candidate child state to be returned (*i.e.*, *next*), we check with the state-signature on Line 33. A *null* state will be returned if *next* is an empty state or a redundant state.

An even better solution is to push the empty-state checking into the *threshold_expand* and *neighborhood_expand*. For *threshold_expand*, we verify each child state before it is inserted into *l_heap* (Line 67, Algorithm 6). However, for *neighborhood_expand*, if a child state *cs* in $N(next)$ is empty, we still need to keep *cs* in the *l_heap*. This is because we may need to further expand to $N(cs)$, which may be non-empty, non-redundant and only

accessible by cs.

A state-signature may be implemented by a bloom filter, which has false positives such that an empty state will be falsely recognized as a non-empty state. Suppose S is a non-leaf empty-state and was falsely passed the signature checking. At certain stage, S may be scheduled for expansion. Since $key(S)$ does not appear in join-signature, the algorithm notices that S is an empty state and the previous false positive will be corrected. For this purpose, we directly return *null* from *S.get_next* (Line 24 of Algorithm 5). Since *S.l_heap* is empty, S will also be discarded by the main query processing loop (Line 13 of Algorithm 5). In this way, the false positives on non-leaf states will not propagate. We have the following lemma with proof omitted.

Lemma 8 In Algorithm 5, the expect number of states retrieved is $(n_I^* - n_{II}^*)fp + n_{II}^*$, where $n_I^* = |\{S|f(S) \leq s^*\}|$ is the type-I optimal state number (Section 5.1.3), $n_{II}^* = |\{S|f(S) \leq s^*$ *and* S *is neither empty nor redundant*$\}|$ is the type-II optimal state number (Section 5.1.3), and fp is the expected false positive rate of the join-signature.

When the original bit-array is used (*i.e.*, $fp = 0$), the number of states retrieved by Algorithm 5 is type-II optimal. Similarly, the expect number of disk accesses for state-signatures (referred as d_s) is $(n_I' - n_{II}')fp + n_{II}'$, where n_I' and n_{II}' are the number of non-leaf states with respect to type-I and type-II optimality, respectively. Since the majority states retrieved are leaf-states, we expect that d_s is far less than n_{II}^*. On the other hand, retrieving a state S does not necessarily incur disk accesses because the index nodes involved in S are shared by other states and may have already been retrieved. Hence, the number of index node accesses (referred as d_i) may also be less than n_{II}^*. In general, we observe $d_s \ll d_i$. Moreover, without state-signature, d_i will be significantly higher (*e.g.*, Table 5.1).

As mentioned in Section 5.3.2, one can use low-dimensional join-signatures to answer high-dimensional queries. Suppose the system pre-computed join-signatures on all pairs of indices. If a query involves $m > 2$ indices, for each state $S = (n_1, n_2, \ldots, n_m)$, we will load

state-signatures of states $S_{ab} = (n_a, n_b)$ (for all $a, b = 1, \ldots, m$ and $a \neq b$). A child state is an empty state if any low-dimensional state-signature returns false. The low-dimensional join-signature may also speed-up child state generation in *threshold_expand* procedure in that whenever a pair of child entries (e_a, e_b) is identified as empty, all child states that are super-sets of (e_a, e_b) can be safely pruned (on Line 64 of Algorithm 6).

5.4 Performance Study

This section reports our experimental results. We compare the query performance among four different methods: the *table scan* (*TS*) approach that sequentially scans the data file and computes top-k; the *baseline* (*BL*) index-merge approach using Algorithm 4; the *progressive expansion* (*PE*) approach with the double heap algorithm only; and the *progressive expansion and join-signature* (*PE+SIG*) approach that applies both double heap algorithm and join-signatures. We first discuss the experimental setting.

5.4.1 Experimental Setting

We use both synthetic and real data sets for the experiments. The real data set is a variation of the *Forest CoverType* data set obtained from the UCI machine learning repository web-site (www.ics.uci.edu/∼mlearn). This data set contains $1,162,024$ data points with 6 selected attributes (cardinalities 255, 207, 185, $1,989$, $5,787$ and $5,827$). We also generate a number of synthetic data sets for our experiments. The *TS* approach sequentially reads tuples from file. In the meanwhile, *TS* maintains a heap with size k to keep track the current top-k results seen so far. For other approaches using index-merge framework, we assume the attributes involved in ranking functions are indexed by either B_+-trees or R-trees. By default, the page size in index nodes is set as $4KB$. All methods are implemented in JAVA.

5.4.2 Experimental Results

We use *execution time* as evaluation metric and conduct experiments to evaluate the query performance with respect to different ranking functions, different type of indices and the number of indices for merging. Guided by the query performance, we further examine how to configure indices for efficient online query processing. Finally, we show the scalability of the proposed methods, including both online query and offline computation costs.

Query Performance w.r.t. Ranking Functions

Since the index-merge paradigm is motivated by supporting non-monotonic ranking functions, we first evaluate query performance with respect to different types of functions. Suppose the ranking function is formulated on two attributes A and B, and each of them is indexed by a B_+-tree. For demonstration, we use three queries with controlled functions: (1) a *semi-monotone* query with function $f_s = (A-a)^2 + (B-b)^2$ where a and b are random parameters. This is a typical nearest neighbor query, which is frequently used in database systems; (2) a *general* query with function $f_g = (A - B^2)^2$. This query is often used to measure the min square error; (3) a *constrained* query with function $f_c = \frac{A+B}{\eta(B)}$, where $\eta(B) = 1$ if $b_1 \leq B \leq b_2$, and $\eta(B) = 0$ otherwise. f_c essentially constraints the value of B to be between b_1 and b_2, and b_1 and b_2 are two random parameters.

We use synthetic data sets in this set of experiments. By default, all data contains $1M$ tuples. The query execution time with respect to different k values and different ranking functions are shown in Figures 5.7 to 5.9. We observe that all the approaches using the index-merge framework perform better than table scan, while the speed-up margin differs from each ranking function: with f_s and f_c, both *PE* and *PE+SIG* are almost one order of magnitude faster than *BL*, which is already one order of magnitude faster than *TS*; and with f_g, *PE+SIG* is 10 times faster than *BL* and *PE*, which are only around 2 times faster than *TS*. The experimental results show the effect of two optimization techniques (progressive merge and selective merge) with respect to different ranking functions.

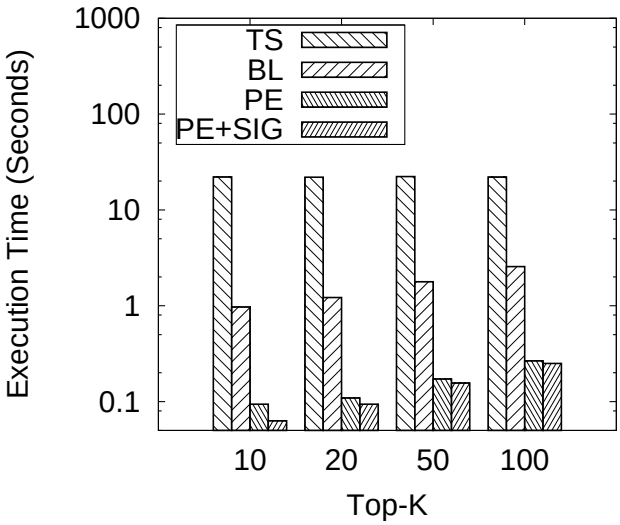


Figure 5.7: **Execution Time w.r.t. K,** $f = f_s$

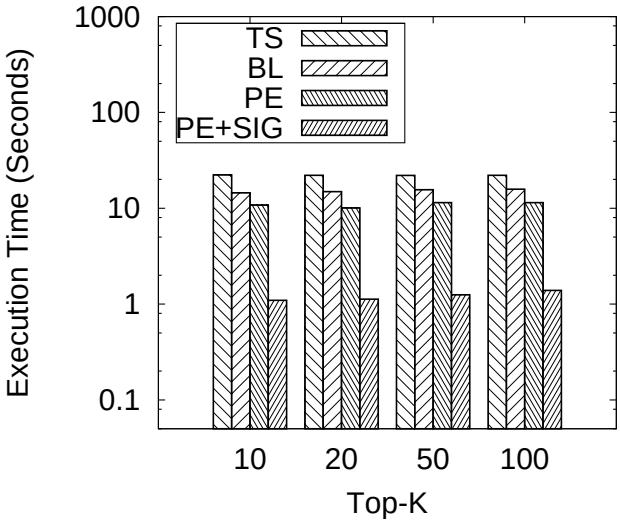


Figure 5.8: **Execution Time w.r.t. K,** $f = f_g$

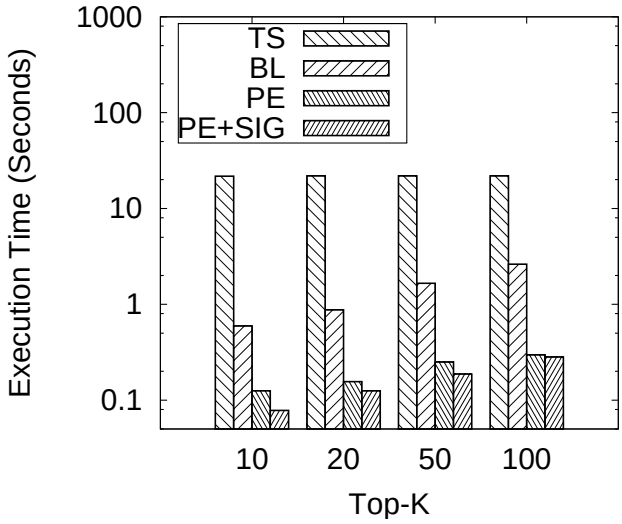


Figure 5.9: **Execution Time w.r.t. K,** $f = f_c$

To explain the difference, we first analyze the properties of the three index-merge based approaches. The overall execution time can be decomposed into two parts: CPU time for state search and state generation; and I/O time for node retrieval. *PE* improves *BL* with respect to CPU cost, and *PE+SIG* further improves *PE* in terms of I/O cost. We then examine the three ranking functions to see which cost (*i.e.*, CPU or I/O) dominates the overall performance. In f_s, the top answers are close to point (a, b), and it is very likely that only a few index nodes need to be retrieved. Thus the I/O may be relatively cheap and the CPU cost dominates. The same observation is also applied on f_c. The hypothesis is well supported by Figures 5.7 and 5.9 in that: (1) *BL* is significantly faster than *TS* because *BL* requests much less I/O; (2) *PE* further improves *BL* by order of magnitude because CPU cost dominates in *BL*; and (3) the speed-up of *PE+SIG* over *PE* is limited because the room for improvement is small (*i.e.*, I/O is already very cheap). On the other hand, f_g is a difficult query function since the top results could scatter over the whole domain, and one has to retrieve more data to answer the query. As the result, the I/O cost dominates. As shown in Figure 5.8, the speed-up of both *BL* and *PE* over *TS* is not significant. *PE+SIG* achieves substantial improvement because the join-signature prunes many empty-states, and

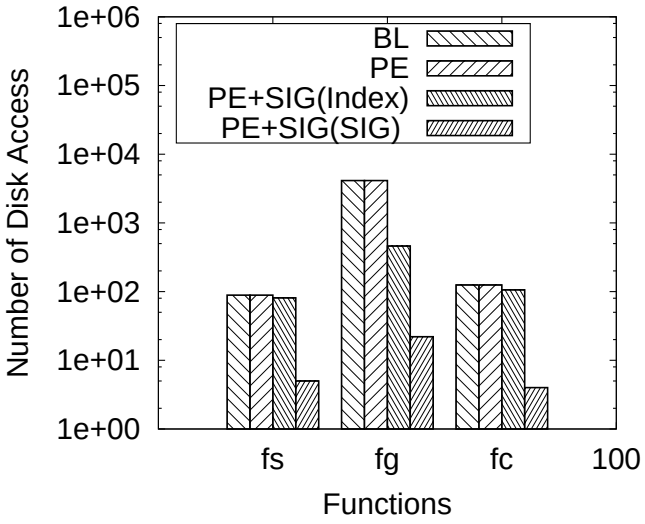


Figure 5.10: **Disk Access w.r.t.** f**,** $k = 100$

thus reduces the I/O requests.

Figure 5.10 shows the number of disk access for three functions when $k = 100$. For *PE+SIG*, we further plot the number of index node requests and that of the state-signature requests. Comparing with the I/O for index nodes, the I/O cost for join-signatures is much less. Among the three functions, f_g incurs most I/O costs, and this is consistent with the above analysis. Figure 5.11 shows the number of generated states. We observe that the progressive expansion is quite effective in that it generates much less states. Sometimes *PE+SIG* generates less states because the pruning of empty states (and whose child states). Finally, Figure 5.12 shows the peak heap size. The heap size of *PE* and *PE+SIG* are computed by the accumulative size of the global heap and all local heaps. Note that even for f_g, the peak heap size of *PE* (*PE+SIG*) is $2,714$ (469) when $k = 100$. This is actually a very important property to support in-memory computation. We will further address this in Section 5.4.2.

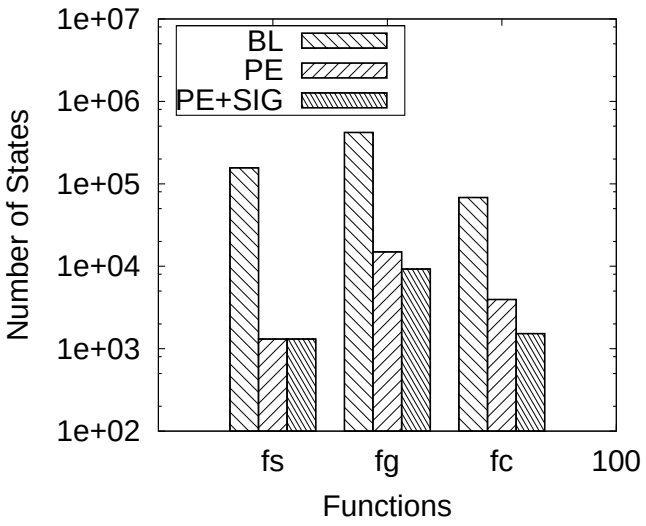


Figure 5.11: **States Generated w.r.t.** f**,** $k = 100$

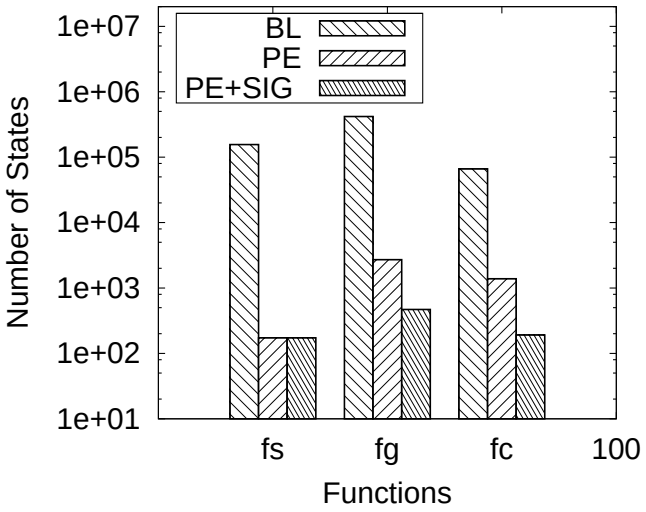


Figure 5.12: **Peak Heap Size w.r.t.** f**,** $k = 100$

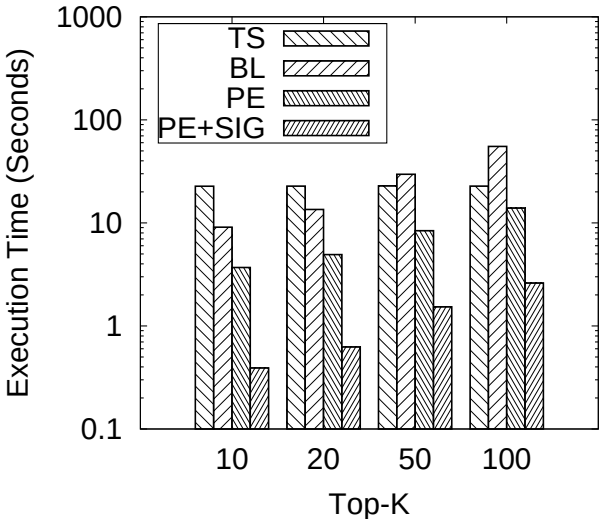


Figure 5.13: **Execution Time w.r.t. K, Real Data**

Query Performance on R-Tree Indices

After reporting the results on B_+-tree indices, we evaluate the query performance on R-tree indices in this subsection. We have demonstrated the differences of three ranking functions in the last subsection and their behaviors are similar with R-tree indices. For simplicity, we only use f_s in the following experiments. Suppose an R-tree index consists of d dimensions. Merging two R-tree indices means there are $2d$ attributes in the ranking function. We define $f_s = \sum_{i=1}^{2d}(A_i - a_i)^2$, where A_i is an attribute value and a_i is the query parameter. It is possible that some attributes are not involved in ranking, and we will address this in Section 5.4.2. As we discussed in Section 5.2, neighborhood expansion is not applicable in R-tree, since the nodes are not fully ordered. We use threshold expansion only.

We first conduct experiments on the real data set, whose 6 attributes are evenly divided into 2 groups. Each group is indexed by an R-tree. We vary the value of k from 10 to 100, and the query execution time is shown in Figure 5.13. Clearly, *PE+SIG* performs best among all approaches. An interesting observation is the *BL* is even worse than *TS* when $k \geq 50$. This is because the ranking function involves 6 attributes, which make it more difficult to search for the final results. To verify this, we generate 4 different data sets with

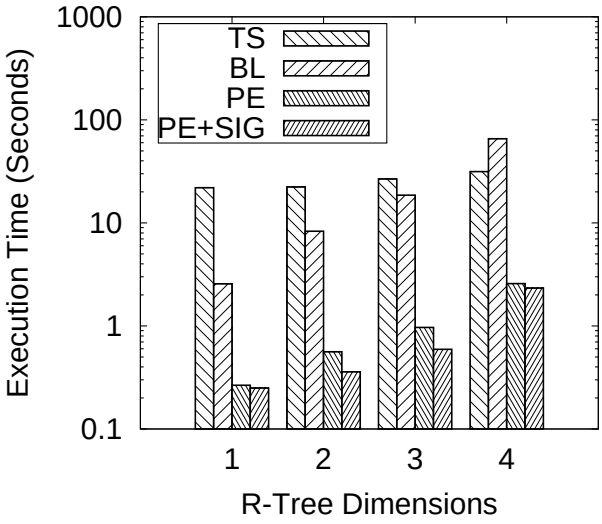


Figure 5.14: **Execution Time w.r.t. R-Tree**

2, 4, 6 and 8 dimensions, and build 2 R-tree indices (by evenly partition the attributes) on each of them. The execution time for $k = 100$ is shown Figure 5.14. As expected, it is more expensive to answer queries with more attributes. However, even for $4d$ R-tree, *PE* (and also *PE+SIG*) finishes query in around 2.5 seconds, which is more than 12 times faster than that by *TS* (31.5 seconds).

Query Performance on 3-Way Merge

All the previous experiments are conducted upon 2-way index merge. Here we examine query performance on 3-way index merge. We use $f_s = \sum_{i=1}^{3}(A_i - a_i)^2$, where A_i are three attributes, and each of which is indexed by a B_+-tree. As discussed in Section 5.3.3, the *PE+SIG* approach has two choices: (1) use one $3d$ join-signature; or (2) use three $2d$ join-signatures (*e.g.*, (A_1, A_2), (A_1, A_3) and (A_2, A_3)). We report query performance for both scenarios.

Figure 5.15 shows the execution time on a synthetic data set with 3 dimensions. We did not report results of *BL*, because *BL* generates too many states and runs out of memory. Although *PE* can effectively control the heap size, its execution time becomes worse than *TS*

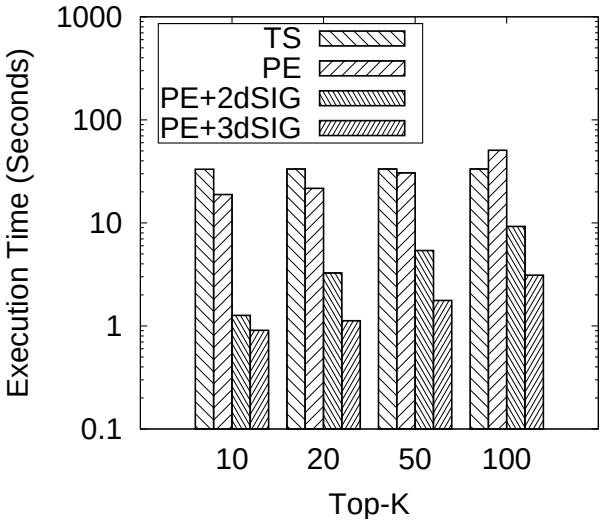


Figure 5.15: **Execution Time w.r.t. K, 3 Indices**

when $k = 100$. Both *PE+SIG* approaches run significantly faster. Particularly, the *PE+SIG* with three $2d$ join-signatures, although not as effective as that with one $3d$ join-signature, performs very well in pruning empty states.

Setting $k = 100$, we plot the peak heap sizes in Figure 5.16 and the number of disk access in Figure 5.17. Clearly, the margin between *PE* and *PE+SIG* becomes much larger comparing with that in 2-way index merge (Figures 5.12 and 5.10). This is because the space of joint state grows exponentially with the number of merging indices. Consequently, both CPU cost for state search and I/O cost for index node retrieval are higher. Moreover, given the large number of candidate states, the probability that a state is not empty drops exponentially. *PE+SIG* achieves significant gain by pruning empty states. We also observe that in both *PE+SIG* approaches, the number of join-signature requests is several times less than that of index node requests.

Index Configuration

Having observed that merging multiple indices introduces high computational complexity in the above subsection, here we discuss how to alleviate the challenges with proper index

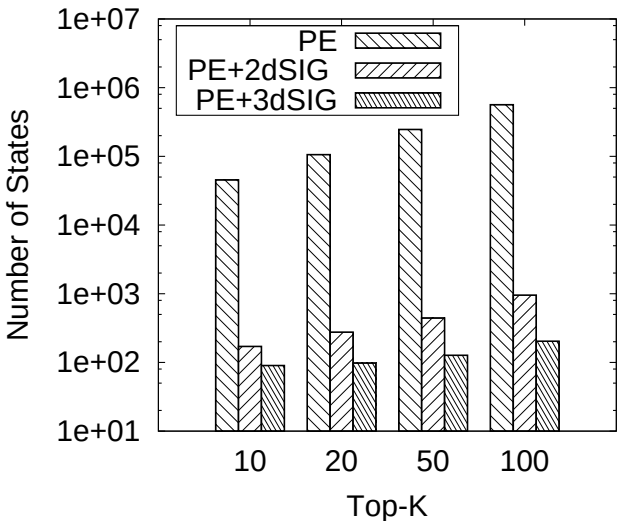


Figure 5.16: **Peak Heap Size w.r.t. K, 3 Indices**

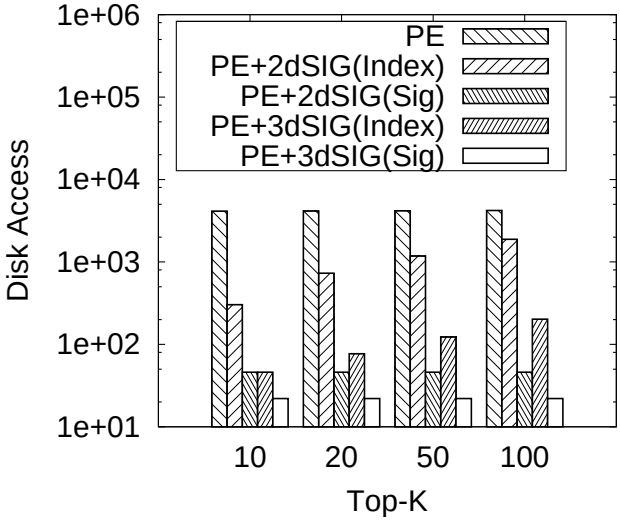


Figure 5.17: **Disk Access w.r.t. K, 3 Indices**

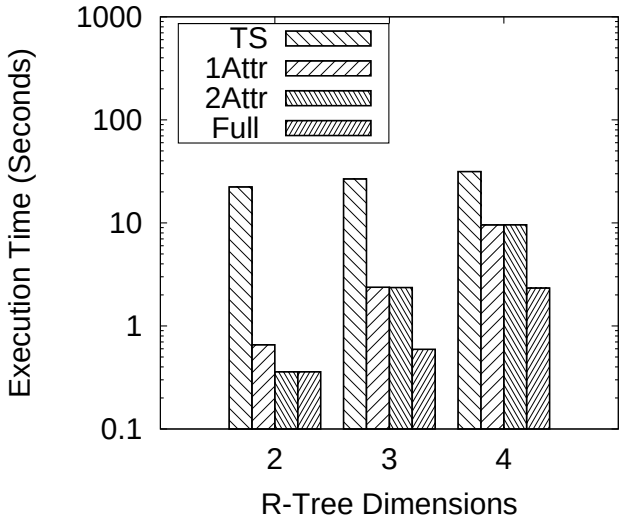


Figure 5.18: **Partial Attributes in Ranking**

configuration. Suppose the database consists m dimensions, and the query attributes are randomly selected. One can build m B_+-trees on each attribute. Alternatively, one can group attributes with size r and build $\frac{m}{r}$ R-trees on each group. For example, when $m = 8$, we can set $r = 4$ and build two R-tree indices.

In Section 5.4.2, we have shown that the ranking functions consisting 8 attributes can be answered fairly efficiently by merging R-tree indices (Figure 5.14). The experiments are conducted under the assumption that all attributes in R-trees are involved in ranking function. In contrast to using low dimensional indices to answer high-dimensional queries, it is interesting to further check whether the high-dimensional indices (*e.g.*, R-tree) can efficiently process low-dimensional queries. We construct three sets of f_s by selecting *one*, *two* and *all* attributes from each R-tree index, and run queries on the same data sets in Figure 5.14. The experimental results of *PE+SIG* (with $k = 100$) are shown in Figure 5.18. Not surprisingly, answering queries with full attributes is the most efficient. However, comparing with *TS*, the performance of partial attributes is still very attractive. For example, when $m = 6$ and $r = 3$, the execution time is around 2.4 seconds for one or two attributes. For the same query, *TS* needs more than 26 seconds.

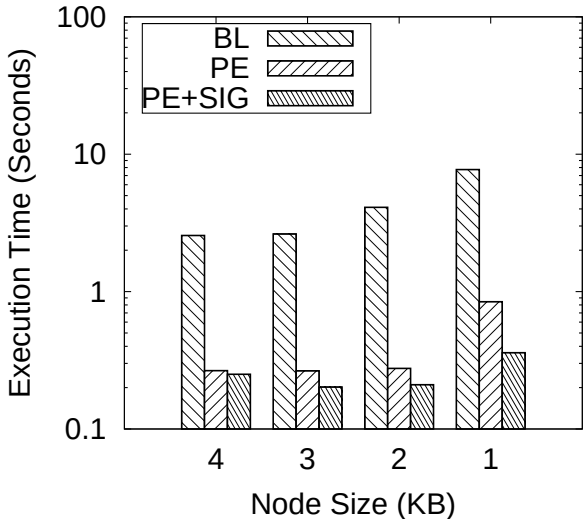


Figure 5.19: **Execution Time w.r.t. Node Size**

For some database with moderate number of ranking attributes (*e.g.*, 4-8), our experiment results suggest that partitioning attributes into two groups and building R-tree index on each of them provides fairly robust query performance for queries involving any subset of attributes.

As part of the index configuration, we also test the query performance by varying the index node size. Typically, the node size is chosen from $1KB$ to $4KB$. We generate a $2d$ synthetic data sets, and build B_+-tree indices with size $1KB$ to $4KB$ on each attribute. The execution time for $k = 100$ is shown in Figure 5.19. With smaller node size, the number of nodes increases, and so does the number of empty-states. On the other hand, with larger node size, the fanout of each node increases, and so does the number of states to be enumerated. In general, *PE+SIG* considers both effects, and thus is not very sensitive to node size.

Scalability

The final set of experiments is to study the scalability of the proposed methods. We use synthetic data set with 2 B_+-tree indices, and vary the number of tuples from $1M$ to $5M$.

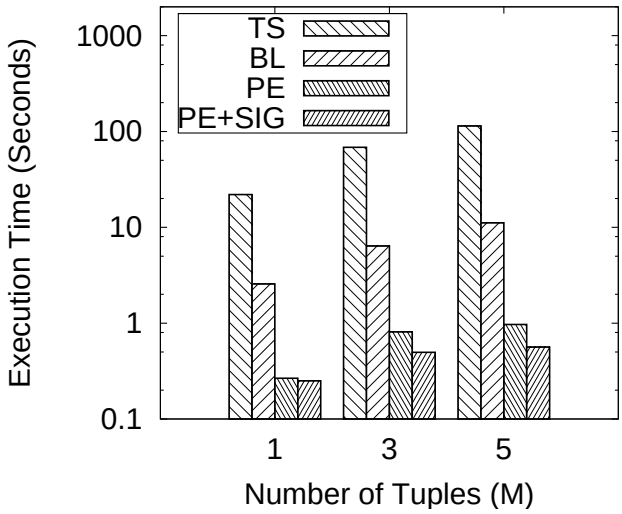


Figure 5.20: **Execution Time w.r.t. T**

The query execution time (with $k = 100$) in Figure 5.20 shows that all methods scale quite well. Besides the online query performance, we also report the construction and space costs for the join-signature in Figures 5.21 and 5.22. To compare with, we plot the query execution time used by *TS* in Figure 5.21, which shows that the join-signature can be computed fairly efficiently in that it is comparable to table scan. We also compare the size of the join-signature with the size of one B_+-tree index in Figure 5.22, and observe that the size of join-signature is at least 6 times smaller.

5.5 Discussion

We discuss the related work and two extensions of the proposed methods: (1) merging indices from multiple relations, and (2) using the index-merge framework to answer other preference queries.

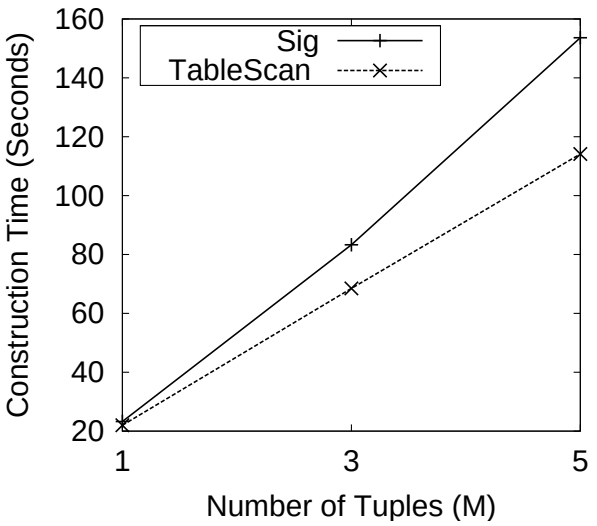


Figure 5.21: **Construction Time w.r.t. T**

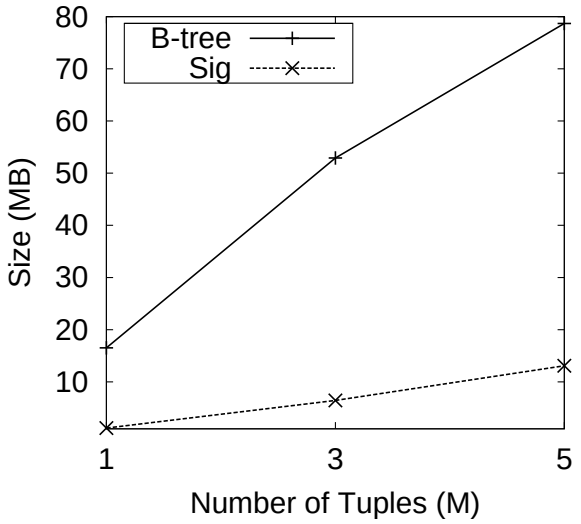


Figure 5.22: **Size of Join-signatures w.r.t. T**

5.5.1 Related Work

Our work on high dimensional data is related to index-based ranked spatial join queries [77, 58], which starts from the roots of indices to be joined and finds pairs of overlapped (or closed) entries. The algorithm is recursively called on the next promising pair which overlaps (or is close) the most, until the top-k results are found. Plane sweep is the popular optimization technique to reduce the search complexity. It applies sorting on one dimension in order to reduce the cost of computing pairs that may belongs to top-k results. Another related work is to compute top-k by interleaf traversal (in merging indices), which starts search from the leaf-nodes containing extreme points and progressively traverses to neighboring leaf-nodes[75].

The join-signature proposed in this chapter is an extension of data cube [27] which pre-computes multi-dimensional aggregates. We treat each index as a dimension, and index nodes as values. With a boolean measure that indicates whether a joint state is empty or not, a join-signature is essentially a data cube over multiple index-dimensions. Materializing join results is also explored by join indices [65]. The join-signature differs from them in that the join-signature is built at the index node granularity, rather than on the data record level.

5.5.2 Merge Indices from Multiple Relations

The methods developed in this chapter can also be used to merge indices from multiple relations. Particularly, we discuss how to extend the method to join primary keys and foreign keys. A more general problem configuration which involves both ranking and join is presented in the next chapter. Assume primary keys and foreign keys are stored together with attribute values in the leaf index nodes. Thus the join condition can be evaluated during index merge.

While the double heap algorithm with progressive expansions can be directly used, there is a small variation to compute the join-signatures. To construct the join-signature of index

A from relation R_1 and index B from relation R_2, we can first compute paths with respect to both indices for each tuple. Suppose each tuple in R_1 is associated with a path $path_A$ and each tuple in R_2 is associated with a path $path_B$. We then conduct a sort-merge join on R_1 and R_2, and keep both $path_A$ and $path_B$ in the join results in R_3. The method presented in Section 5.3.2 can be applied on R_3 to compute the join-signature.

5.5.3 General Preference Queries

Top-k queries are related to several other preference queries, such as skyline query [12] and convex hulls [11]. Skyline query asks for the objects that are not dominated by any other object in all dimensions. A convex hull query searches a set of points that forms a convex hull of all the other data objects. The methodology developed in this chapter is also applicable to these queries. The key observation is that all the queries can be processed progressively in the top-down fashion. For demonstration, we discuss how to apply our method for skyline computation as follows. The method for convex hull queries is similar.

In [51], Papadias *et al.* developed a branch-and-bound search algorithm that progressively retrieves R-tree nodes, from root to leaves, until all the skylines are found. At any stage, if a nodes n is dominated by a data object, all the child nodes of n can be pruned. The same rule can be applied on the state space in this chapter: If a state S is dominated by a data object, all the child states of S can be pruned. As soon as all the states in the global heap are pruned, the search for the skyline objects halts. The join-signature can be used without any modification, since the empty-states do not contribute to the final results either.

The progressive expansion methods are also applicable in skyline computation. Let $\overline{n}$ and $\underline{n}$ be the maximal and minimal attribute values among the region covered by n. Given a state $S = (A_1, B_1)$, with child nodes $(a_1, a_2, \ldots, a_n)$ and $(b_1, b_2, \ldots, b_m)$, we sort a_i and b_i according to $\underline{a_i}$ and $\underline{b_i}$, respectively. Similar to the threshold expansion in Section 5.2.3, we can progressively generate child states until a threshold position (r, t), such that there exists a generated child state $c = (a^*, b^*)$ satisfying $\overline{a^*} \leq \underline{a_r}$ and $\overline{b^*} \leq \underline{b_t}$. Consequently, the

$S.get\_next$ method returns all the generated child states (instead of one child state in top-k query) and updates S's coordinate values as $\underline{a_r}$ and $\underline{b_t}$.

Algorithm 5 Progressive and Selective Merge

Input: A set of Indices I, ranking function f, top-k

1: $TopK = \phi$; //heap with size k to hold current top-k
2: $g\_heap = \{Joint\ Root\}$; //heap for state search
3: **while** ($g\_heap \neq \phi$ *and* $f(TopK.root) > f(g\_heap.root)$)
4: remove top entry S from g_heap;
5: **if** (S is empty or redundant)
6: **continue**;
7: **if** (S is a leaf state)
8: Retrieve data and update $TopK$;
9: **else** //S is a non-leaf joint state
10: $next = S.get\_next()$
11: **if** ($next \neq null$) // non-empty non-redundant
12: insert $next$ to g_heap;
13: **if** ($S.l\_heap \neq \phi$) // more child states in S
14: insert S to g_heap;
15: **return**

Procedure $S.get\_next()$
Vars: Local heap: l_heap

21: **if** ($l\_heap = \phi$)// the first time $S.get\_next$ is called
22: load state-signature sig;
23: **if** ($sig = null$) //no signature for empty-states
24: **return** null; //return nothing, keep $l\_heap = \phi$
25: load index nodes of S;
26: **if** (f is (semi-)monotone in S); //neighborhood
27: $next = neighborhood\_expand()$;
28: **else** //threshold
29: $next = threshold\_expand()$; //the best child state
30: **if** ($l\_heap \neq \phi$) //there are child states left
31: $f(S) = f(l\_heap.root)$; //l_heap was updated
32: **if** ($next$ is empty or redundant)
33: $next = null$;
34: **return** $next$;

Algorithm 6 Neighborhood and Threshold Expansions

Procedure *neighborhood_expand*()
41: **if** ($l\_heap = \phi$)// the first time *S.get_next* is called
42: $l\_heap = I(S)$;// insert initial states
43: remove top entry *next* from *l_heap*;
44: insert $N(next)$ to *l_heap*;
45: **return** *next*;

Procedure *threshold_expand*()
Vars: $S = (n_1, \ldots, n_m)$, and $n_i = (e_i^1, \ldots, e_i^{M_i})$ for each i
Current threshold position t_i $(i = 1, \ldots, m)$

51: **if** ($l\_heap = \phi$)// the first time *S.get_next* is called
52: $l\_heap = (e_1^1, \ldots, e_m^1)$; // insert initial states
53: $t_1 = \ldots = t_m = 2$; //initial threshold positions
54: $find\_next()$; //find the first candidate
55: remove top entry *next* from *l_heap*;
56: $find\_next()$;//search for next state
57: **return** *next*

Procedure $find\_next()$
61: **while** ($f(l\_heap.root) > \min(f'(e_1^{t_1}), \ldots, f'(e_m^{t_m}))$
62: and $(\exists i \in \{1, \ldots, m\}$ such that $(t_i \leq M_i)))$
63: $s = \arg\min_{i=1}^{j} f'(e_i^{t_i})$;
64: $news = [e_1^1, \ldots, e_1^{t_1 - 1}] \times \ldots [e_s^{t_s}] \ldots [e_j^1, \ldots, e_m^{t_m - 1}]$;
65: t_s++;
66: **for** each *cs* in *news*
67: **if** (*cs* is not empty or redundant)
68: insert *cs* to *l_heap*;
69: **return**;

Chapter 6

Ranking with Joins

6.1 SPJR Queries

Many ranking queries in the context of relational database are the *SPJR* (*i.e.*, Selection, Projection, Join and Ranking) queries, whose boolean predicates consist of both selections and joins. Given a *SPJR* query, the current DBMS first generates the complete results according to the boolean query constraints, then sort the results and report the top-k answers. This approach is often inefficient in that joins may generate a huge number of output. To efficiently processing ranked queries, the query evaluation system should be able to identify and only touch the *subset of joined results* that is most promising for top-k answers. This is found to be more difficult than querying over single relation since the joined results are generated on-the-fly.

6.1.1 Query Model

Consider m relations $R_1, R_2, \ldots, R_m$, and each R_i has a set of attributes $\mathcal{A}_i = \{A_i^1, A_i^2, , \ldots\}$. We assume the ranking functions will only involve a subset of pre-determined *ranking attributes*: $\mathcal{F}_i = \{N_i^1, N_i^2, \ldots\} \subseteq \mathcal{A}_i$. An *SPJR* query specifies the boolean query constraints on a subset of non-ranking attributes and formulates a ranking function on a subset of ranking attributes. The result of a top-k query is an ordered set of k tuples that is filtered by the boolean constraints and ordered according to the given ranking function. A possible SQL-like notation for expressing *SPJR* queries is as follows:

SELECT	*Top* k $\mathcal{P}$
FROM	$R_1, R_2, \ldots, R_m$
WHERE	$b(\mathcal{A}')$
ORDER BY	$f(\mathcal{F}')$

Here, $\mathcal{P} \subseteq \cup_{i=1}^{m} \mathcal{A}_i$ is a subset of the projection attributes; b is a boolean predicate and $\mathcal{A}' \subseteq \cup_{i=1}^{m} \mathcal{A}_i$ is a subset of any attributes; and f is a ranking function and $\mathcal{F}' \subseteq \cup_{i=1}^{m} \mathcal{F}_i$ is a subset of the ranking attributes. Without loss of generality, we assume the ranking has minimization preference. An example of an *SPJR* query is shown as follows.

Example 7 *Consider a user who wants to fly from Chicago to a coastal city and stay in a* 3*-star hotel. Given a flight relation* F *and hotel relation* H*, the user may issue the following query to search for a plan which minimizes the combined flight and hotel costs.*

SELECT	*Top 10* H.CityName, H.HotelName
FROM	F, H
WHERE	F.Departure = "Chicago" *and*
	F.Destination = H.CityName *and*
	H.In_Coastal_City = true *and*
	H.StarLevel = 3
ORDER BY	F.Price + H.Price *asc*

The task of searching for the interesting data regions relies on the analysis of the ranking function, which is a typical problem in numerical analysis. However, most ranking functions in real life applications have nice properties, which can be utilized by the query optimizer. For simplicity, we demonstrate our query optimizer by two commonly used ranking functions: the *convex* functions and the *monotone* functions.

6.1.2 System Architecture

We extend the original ranking cube framework to multiple relations (*i.e.*, multiple ranking cubes). Furthermore, we propose a complete system to support *SPJR* queries. The system is built on top of ranking cubes, and consists of a *query optimizer* and a *query executer*. An

overview of the architecture is presented in Figure 6.1, with the major features highlighted as follows.

Query Optimizer: The goal of the query optimizer is to minimize the retrieval of data in answering *SPJR* queries. The query optimizer determines whether the system should continue to retrieve data and what data needs to be retrieved. Our query optimizer differs from the traditional cost-based optimizer [56] in two ways. First, current query optimizers try to select the best access path for a given query. We argue that in most ranked queries, all of the *related* data can be loaded into memory. Hence the access path will not play a critical role in query optimization. This is because only a small portion of the data which may contribute to the final top-k answers need to reside in memory, whereas most of the data is neither retrieved nor thrown away after being determined as non-contributing to answers. Second, most current query optimizers are static in the sense that the query plan is determined before the query is executed. Instead of estimating the size of the interesting tuples beforehand, it is much easier for an SPJR query optimizer to find the promising data regions dynamically and progressively. As a result, our query optimizer is also involved during the query execution, and the query plan is dynamic.

Query Executer: The query executer consists a set of operators, which includes *Selection* (*i.e.*, select the *tid* list from ranking cubes); *Join* (*i.e.*, join multiple *tid* lists); *Retrieve* (*i.e.*, retrieve the candidate tuples from original relations) and *Sort* (*i.e.*, sort the candidate results by ranking functions). The query executer can be summarized as a two-phase approach. In the candidate-generation phase, the query executer progressively selects *tid* lists from ranking cubes, according to the instruction from the query optimizer. The selected *tid* lists are then evaluated by the join operator. The candidates generated by this phase are then feeded into the second phase, the candidate verification phase , where the values of the candidates are retrieved from the base relations and their rankings are evaluated.

The remainder of this chapter is organized as follows. We discuss the query optimizer in Section 6.2, and the query executer in Section 6.3. Section 6.4 presents the performance

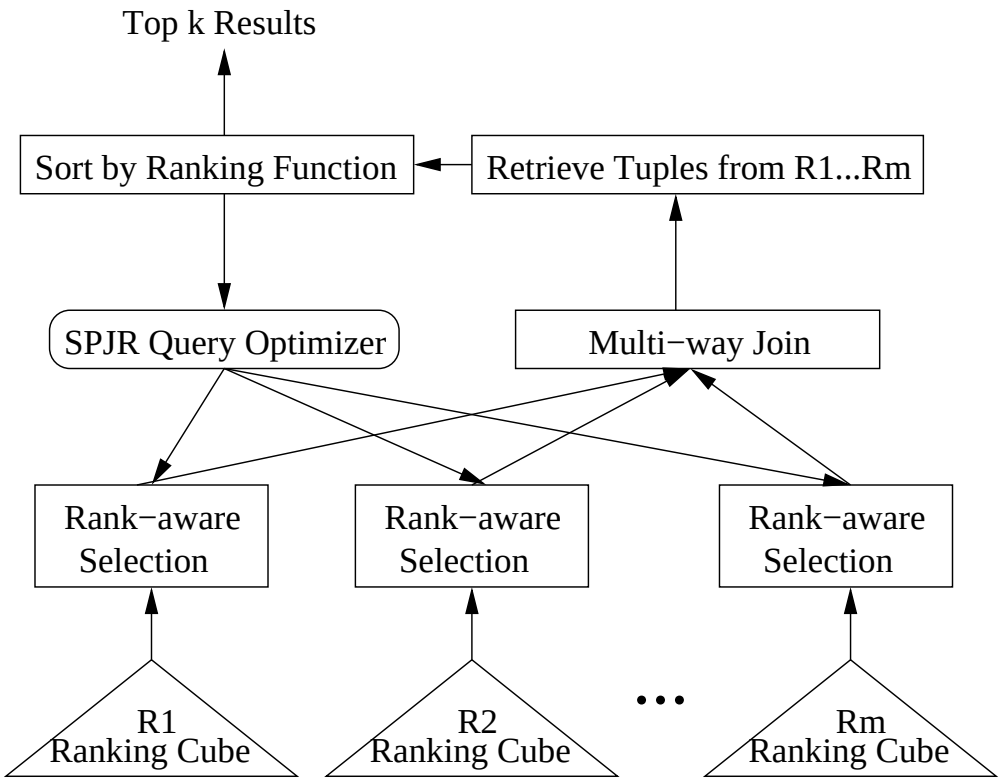


Figure 6.1: **An overview of Ranking Cube System**

study.

6.1.3 Ranking Cube for Join

The ranking cube is built on the non-ranking attributes and the block attribute B. Each cuboid in the ranking cube is named by the involved attributes. For example, the cuboid AB corresponds to non-ranking attributes (or selection dimension) A and the block attribute B. Each cell in a cuboid consists of a cell identifier (*e.g.*, $A = a_1, B = b_1$) and a list of corresponding *tid*s (*e.g.*, $\{t_1, t_4\}$). An multi-dimensional index is built on the attributes to efficiently access the *tid* lists. There are two typical ordering of attributes in the multi-dimensional index: B in head and B in tail.

First, B is the first attribute and some non-ranking attributes are arranged after B in the index. When a *tid* list is selected by the value of B, they are grouped by the values on

the non-ranking attributes. This data organization is typically useful for evaluating joins and this type of cuboid is called *join cuboid*. Second, when B is the last attribute and some non-ranking attributes are arranged before B in the index, the cuboid can be used to answer data request with boolean selection conditions on those non-ranking attributes. We call this type of cuboid as *selection cuboid*. In selection cuboid, *tid* lists are selected by a specified B value and other boolean selection conditions, the size of *tid* list is often much smaller (sometimes empty) than the expected block size P. To avoid issuing multiple random access, we discusses the *pseudo block* in Chapter 3, where nearby blocks are grouped and retrieved together.

6.2 Query Optimizer

In this section, we examine the query optimizer based on the ranking cube model (in Chapter 3). The base unit in query optimization is block. The query optimizer progressively instructs the query executer which data block to be retrieved. A data retrieval request on a ranking cube consists of three components: (1) $\mathcal{S}$, a boolean selection predicate (*i.e.*, from the boolean query constraint), (2) $\mathcal{J}$, a set of join attributes (*i.e.*, the attributes used in Join), and (3) $\mathcal{B}$, a target block ID. Ex. 8 shows a sample request.

Example 8 *Following Ex. 7, the ranking cubes of relation H and F are built. A data retrieval request on the Ranking Cube of F could be:* $\mathcal{S}$ *=(Departure="Chicago");* $\mathcal{J} = \{Destination\}$ *and* $\mathcal{B} = b_1$.

While both $\mathcal{S}$ and $\mathcal{J}$ are determined from the boolean query constraint, the value of $\mathcal{B}$ is dynamic during query execution. $\mathcal{B} = *$ means all blocks are requested (*i.e.*, the ranking function does not have attributes from this ranking cube). In the following, we discuss how to progressively search for $\mathcal{B}$, and what is the stop condition for query execution. We first review the method on one relation, and then extend it to multiple relations.

6.2.1 Query Optimization over Single Relation

A block is *necessary* to be retrieved if and only if without retrieving this block, the system is not able to claim that the top-k results are found. The query optimizer makes one block request at a time, and the data retrieved from the ranking cube is first evaluated by the boolean query constraints and then sorted by the ranking function. Suppose the current top k^{th} result has score v_k. If there is no block which may contain a data tuple whose score is less than v_k, then it is safe to claim the top-k results are found and the query execution can be stopped. Otherwise, it is necessary to retrieve more data for further evaluation. Given a block b, let $v(b)$ be the possibly minimal score in the region covered by b. The following claim says the optimal choice is to get the block with minimal $v(b)$.

Claim 1 *Let L be the set of blocks already retrieved, if $v_k < min_{\forall b \notin L} v(b)$, $\mathcal{B} = argmin_{\forall b \notin L} v(b)$.*

In most cases, the ranking function is convex or monotone, we can easily find the extreme point which has the globally minimal score, and $\mathcal{B}$ is the block containing this point. When there are some blocks already retrieved, the following claim shows how to reduce the search space.

Claim 2 *If the ranking function is convex or monotone, $\mathcal{B} = argmin_{\{\forall b | N(b,b'), b \notin L, b' \in L\}} v(b)$, where $L \neq \phi$ is the set of blocks already retrieved, $N(b,b') = true$ if and only if b and b' are neighboring blocks.*

The method can be generalized to handling *ad hoc* ranking functions. The basic idea is to decompose the whole domain of the function variables into multiple sub-domains so that in each sub-domain, the function has convex property. The candidate blocks in each sub-domain then compete for the global next block to be retrieved.

6.2.2 Query Optimization on Multiple Relations

For multiple relational queries, the optimizer works on a joint space of blocks from each involved ranking cube. A joint block can be written as $l = (b_1, b_2, \ldots, b_m)$, where $b_1, b_2, \ldots, b_m$ are the blocks from the m involved ranking cubes. To clarify, we call the joint block space as *logical block*, and the block spaces in individual ranking cubes as *physical blocks*.

Claim 1 still holds in the logical block space. To execute a query, the optimizer needs to figure out which physical block to be retrieved. Before we generalize Claim 2 to multiple relations, we first introduce some terminologies in terms of logical block space. Let $v(l)$ be the possibly minimal score of the ranking function in the joint block space. We say a logical block is *retrieved* if all involved physical blocks are retrieved. Two logical blocks l and l' are *neighboring blocks* (*i.e.*, $N(l, l') = true$) if and only if there is only one physical block in l differs from that in l', and the differing blocks are neighboring physical blocks in the corresponding ranking cube. Suppose the differing physical blocks are b_i and b'_i. We denote $D(l|l') = b_i$. Below is the extension of Claim 2 to the logical block space. Ex. 9 demonstrates the procedure of query optimization on multiple relations.

Claim 3 *If the ranking function is convex or monotone, and* $(l^*, l'^*) = argmin_{\{\forall(l,l')|N(l,l'),l\notin L,l'\in L\}} v(l)$, *then* $\mathcal{B} = D(l^*|l'^*)$, *where* $L \neq \phi$ *is the set of logical blocks already retrieved,* $v(l)$, $N(l, l')$ *and* $D(l^*|l'^*)$ *are defined above.*

Example 9 *Table 6.1 shows two relations* R_1 *(with ranking attributes* N_1 *and* N_2*) and* R_2 *(with ranking attributes* M_1 *and* M_2*). Suppose the range of all ranking attributes is* $[0, 1]$. *The bin boundaries used in partitioning* R_1 *are* $Bin_{N1} = [0.4, 0.45, 0.8]$ *and* $Bin_{N2} = [0.2, 0.45, 0.9]$, *and those used in partitioning* R_2 *are* $Bin_{M1} = [0.25\ ,0.7, 0.8]$ *and* $Bin_{M2} = [0.4, 0.5, 0.7]$ *(Fig. 6.2).*

Suppose the query has a join condition $R_1.A = R_2.B$, *and a ranking function* $f = R_1.N_1 + R_1.N_2 + R_2.M_1 + R_2.M_2$. *Top* 2 *results are requested. The globally minimal value of* f *is* 0 *(by* $R_1.N_1 = R_1.N_2 = R_2.M_1 = R_2.M_2 = 0$*). The optimizer locates the first logical*

tid	A	N_1	N_2
t_1	1	0.05	0.05
t_3	2	0.05	0.25
t_4	1	0.35	0.15
...	...	...	...

tid	B	M_1	M_2
s_1	1	0.05	0.15
s_2	2	0.02	0.05
s_5	1	0.30	0.05
...	...	...	...

Table 6.1: **Relations R_1 (left) and R_2 (right)**

block as $l_1 = (R_1.b_1, R_2.b_1)$ (Fig. 6.2), and retrieves $R_1.t_1$, $R_1.t_4$, $R_2.s_1$ and $R_2.s_2$. Both $R_1.t_1$ and $R_1.t_4$ can join with $R_2.s_1$, and their ranking scores are 0.3 *and* 0.6*, respectively. $R_2.s_2$ cannot pass the join condition at this stage, thus are not considered. There are four neighboring logical blocks of l_1: $(R_1.b_1, R_2.b_2)$, $(R_1.b_1, R_2.b_5)$, $(R_1.b_2, R_2.b_1)$ and $(R_1.b_5$, $R_2.b_1)$. The logical block $l = (R_1.b_5, R_2.b_1)$ has the best possible score $v(l) = 0.2$, which is less than the current top 2^{nd} candidate score* 0.6*. We continue to retrieve the physical block $R_1.b_5$. $R_1.t_3$ is retrieved and joins with $R_2.s_2$. The new candidate $(R_1.t_3, R_2.s_2)$ has score* 0.37*. The system continues to look for the neighboring logical blocks. However, none of them has less score than* 0.37*. The top-*2 *results (i.e., $(R_1.t_1, R_2.s_1)$, $(R_1.t_3, R_2.s_2)$) are found, and the query execution stops.*

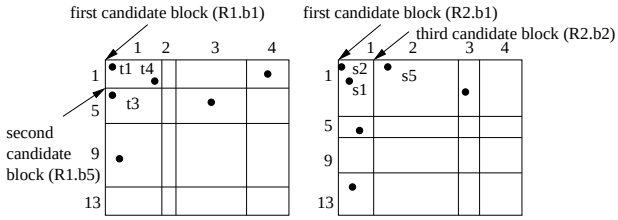


Figure 6.2: **Processing a top-**2 **query with 2 relations**

6.2.3 Comments on Query Optimizer

Our query optimizer is an extension of the sort-merge framework [28, 30] in the middle-ware scenario. The ranking cube can be considered as a data source in their framework. However, the original data source is a sorted list (thus there is only one way to access the data), and only monotone functions are applicable. Our system can support more general ranking

functions. Moreover, the block-level design provides more flexible data access, and the query optimizer addresses an important optimization problem: what is the optimal way to retrieve data blocks?

We use convex and monotone functions to demonstrate the query optimization. The framework can be generalized to handling *ad hoc* ranking functions. The basic idea is to decompose the whole domain of the function variables into multiple sub-domains so that in each sub-domain, the function has convex property. The candidate blocks in each sub-domain then competent for the global next block to be retrieved.

6.3 Query Executer

This section discusses the operators in the query executer. As shown in Fig. 6.1, there are four operators: *Rank-aware Selection*, *Multi-way Join*, *Retrieve tuples from* $R_1, \ldots, R_m$ and *Sort by Ranking Function*. The first two operators manipulate *tid* lists, and generate candidates which satisfy the boolean query constraint. Each candidate is a set of *tid*s from different relations. Given those candidates, the *retrieve* operator uses those *tid*s to get the tuples from the original relations and the *sort* operator evaluates the ranking function with each candidate and discard those candidates whose scores are less than v_k (*i.e.*, the current top k^{th} score). In the following, we present the detailed implementations of the *selection* and *join* operators.

6.3.1 Rank-aware Selection

As discussed in Section 6.2, a selection request issued by the query optimizer consists of a triple: ($\mathcal{S}$, $\mathcal{J}$, $\mathcal{B}$). To answer the data request, there are three steps in the selection operator.

First, using the attributes involved in $\mathcal{S}$ and $\mathcal{J}$, the query executer determines which cuboid to be used: an attribute in $\mathcal{S}$ links to a selection cuboid, and an attribute in $\mathcal{J}$ links to a join cuboid.

Example 10 *Continue on Ex. 8. The ranking cube of the flight relation F has cuboids $A_{Departure}B$, $A_{Destination}B$, $BA_{Departure}$ and $BA_{Destination}$. Here $A_{Departure}$ and $A_{Destination}$ represent the attributes of departure city and destination city in F, and B is the block attribute created by ranking cube. Given a data request ($\mathcal{S}$ = (Departure="Chicago"), $\mathcal{J}$ ={Destination}, $\mathcal{B} = b$), the query executer will select the selection cuboid $A_{Departure}B$ and join cuboid $BA_{Destination}$.*

Second, in order to maximize the benefit of block-level access, we group neighboring cells in selection cuboids, and those cells will be retrieved together. Suppose in cuboid $A_{Departure}B$, cells $c = (\text{“}Chicago'', b)$, $c_1 = (\text{“}Chica\ go'', b_1)$ and $c_2 = (\text{“}Chicago'', b_2)$ are grouped together. The selection operator retrieves all three cells from disk at the same time. Cell c is returned by the request, and cells c_1, c_2 are buffered for the future requests.

Third, an ideal cuboid to answer the data request in Ex. 10 is $A_{Departure}BA_{Destination}$. Since this cuboid was not materialized, we will need to compute the requested cell online. We first retrieve a *tid* list l_1 from $A_{Departure}B$ cuboid using $A_{Departure} = \text{“}Chicago''$ and $B = b$; then retrieve a *tid* list l_2 from $BA_{Destination}$ cuboid using $B = b$. The two lists l_1 and l_2 are intersected, and the result is returned.

6.3.2 Multi-way Join

After the *tid* lists are selected from the ranking cube, the query executer uses a multi-way join operator to evaluate the join constraints. Before we proceed to the details of the join operator, we characterize two types of *tid* lists. A *tid* list is associated with a data request $(\mathcal{S}, \mathcal{J}, \mathcal{B})$. We call a *tid* list as *ranked list* if $\mathcal{B} \neq *$, otherwise, the *tid* list is a *non-ranked list*. The ranked lists are typically short since it is selected from a particular block, while the non-ranked list could have large size. According to the input list, we discuss two implementations of the join operator: *list merge* and *index search*.

List Merge: When join is applied on ranked lists (or short non-ranked list), we imple-

ment the join by list merging. This approach retrieves all lists from the ranking cube, and validate the join conditions in memory. Candidates passed the join condition will be fed to the sort operator. The *tid* lists which fail to pass the join filter may or may not need to be kept in memory (Section 6.3.3). This is because our data selection has ranking preference and is progressive. It is possible that some lists retrieved in future can join with the current lists. To save the space and computational costs, we can use join indices [65] in the physical implementation, as shown in the following example.

Example 11 *Follow Ex. 7, the rank-aware selections return two tid lists from the ranking cubes of H and F, respectively. The list from F ranking cube (i.e., l_F) is indexed by $A_{Destination}$, and the list from H ranking cube (i.e., l_H) is indexed by $A_{CityName}$. Suppose l_F = {("Hawaii", $list_1$), ("Miami", $list_2$)}, and l_H = {("Hawaii", $list_3$), ("Cancun", $list_4$)}. The join index with respect to $F.A_{Destination} = H.A_{CityName}$ is l_{FH} = {("Hawaii", $list_1$, $list_3$)}.*

Index Search: Sometimes a non-ranked list could have large size. Instead of retrieving the whole list, which might be expensive, we can use index search on the non-ranked list. One scenario is demonstrated in Ex. 12.

Example 12 *Follow Ex. 7. Suppose a user only looks for cheapest flight tickets to a coastal city, and the query is:* select top 10 F.Price from F and H where F.Destination=H.CityName and H.In_Costal_City=true*, the query executer needs to join a ranked list from F and a non-ranked list from H. An index search approach retrieves the ranked list from F, and issue multiple index search on H to verify whether a destination city is a coastal city.*

Each index search incurs a random access, which is also not trivial. It is often beneficial to identify the misses (*i.e.*, the index search fails) beforehand. A possible solution is to precompute a signature of the keys in the non-ranked list, and only search those keys confirmed by the signature (without false negative). A well-known method for this purpose is the Bloom Filter [10]. Retrieving the signature is far more efficient than the original list.

To decide whether *to Merge or to Search*, we can use a standard cost based solution. Let the average cost of one random access be C_r, and the average unit cost of retrieving a list be C_l. Suppose the join selectivity of join in a top-k query is s and the length of the list is L. The cost of index search is $\frac{kC_r}{sL}$, and the cost of list merge is LC_l. The value of L can be maintained by ranking cube and only requires one random access. C_r and C_l can be determined by experiments. The estimation of s is addressed by some previous works, such as [13].

6.3.3 List Pruning

Our query evaluation assumes that all lists retrieved so far can fit in memory. An important issue for memory management is to identify the subset of retrieved *tid*s which are not able to contribute for top results (*i.e.*, do not exist in final answers). Removing these *tid*s from memory can reduce not only the memory requirement, but also the computational cost of join operator.

For simplicity, we use a two-way join as an example to demonstrate how to determine that a *tid* is safe to be discarded. Suppose the k^{th} best score on the current joined candidates is v_k. The best possible score that a *tid* can generate can be computed by the block where the *tid* resides and the best untouched block in the other ranking cube. Searching for the best untouched block in the ranking cube is similar to our optimizing method discussed in Section 6.2.2, and we omit the details here.

6.4 Performance Study

6.4.1 Experimental Setting

We compare the performance of ranking cube with the *baseline* solution, where we directly issue the query to Microsoft SQL-Server 2005. To test the performance with different pa-

rameters, we use the synthetic data sets. Given a relation with m non-ranking attributes $A_1, \ldots, A_m$, we create m selection cuboids A_iB and m join cuboids BA_i (see Section 6.2.1). For a fair comparison, we load all cuboids into SQL Server, and build a clustered index for each cuboid. For the baseline approach, we build an index on each non-ranking attribute. We use a simple *SPJR* query as shown below for performance evaluation.

Select Top 10 $R_1.id,\ R_2.id$ From R_1, R_2
Where $R_1.A_1 = 1$ *and* $R_2.A_1 = 1$ *and* $R_1.A_2 = R_2.A_2$
Order By $R_1.N_1 + R_1.N_2 + R_2.N_1 + R_2.N_2$

6.4.2 Experimental Results

The synthetic relations R_1 and R_2 have two ranking attributes (N_1, N_2) and two non-ranking attributes (A_1, A_2). To test the effect of boolean selectivity, we fix the number of tuples as $1M$ and vary the cardinality of each non-ranking attributes from 10 to 30. The query evaluation time using both methods is shown in Fig. 6.3. We then fix the cardinality as 30 and vary the number of tuples in each relation from $1M$ to $5M$. The experimental results are shown in Fig. 6.4. We observe that our method is 1-2 orders of magnitudes faster than the baseline approach. The performance of the baseline approach becomes better when the query is more selective. When we increase the database size, we see a clear trend that the time used by our method is fairly robust, while the time used by baseline approach grows exponentially.

In terms of the space usage, when a relation has $5M$ tuples (with cardinality 30), the base table consumes $143MB$; the indices built on the base table consumes $328MB$; and the ranking cube consumes $606MB$. The overall space requirement of our system is $\frac{606+143}{328+143} = 1.6$ times that of the baseline approach. This is a fairly acceptable cost paid for materialization since the online query processing becomes much more efficient. Furthermore, since we store

Figure 6.3: **Execution Time w.r.t. Cardinalities**

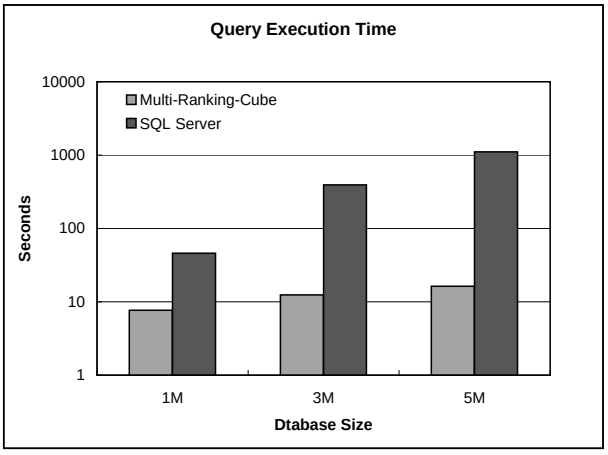


Figure 6.4: **Query Execution w.r.t. Database Size**

ranking fragments in relational database, a large portion of the space is used to store the cell identifiers. The space requirement can be further reduced if we store the data out of the relational database.

Chapter 7

General Preference Queries

7.1 Querying Skylines with Boolean Predicates

7.1.1 Introduction

The preference queries have been studied in the context of skyline query [12, 26, 42, 51, 62], top-k query, convex hull query, *etc.* [12, 46]. In this chapter we focus on skyline queries, however, the developed methodology applies to other types of preference queries as well. Given a set of n objects $p_1, p_2, \ldots, p_n$, a *skyline query* returns all the objects p_i such that p_i is not dominated by any other object p_j. Let the value of p_i on dimension d be $v(p_i, d)$. We say p_i is *dominated by* p_j if and only if for each preference dimension d, $v(p_j, d) \leq v(p_i, d)$, and there is at least one d where the equality does not hold. We refer the results computed by this criterion as *static skylines* since the value of $v(p_i, d)$ is fixed and does not depend on any specific query. A more powerful criterion, called *dynamic skyline* [51], conducts the domination analysis through user-defined mapping functions. Suppose $f_1, f_2, \ldots, f_m$ are m mapping functions, each of which takes an object as input and generates a real value as output. The m functions map each object into a new m-dimensional space. A dynamic skyline is the set of all objects p_i such that p_i is not dominated by any other object p_j in the transformed space.

We address the problem of *efficient processing dynamic skyline queries with multi-dimensional boolean predicates, i.e., OLAPing dynamic skylines*. In the following, we give the motivating examples and the problem formulation.

7.1.2 Examples

One application scenario is to support skyline query based on user-specified preferences in any subset of data (Ex. 1). Another is to compare skylines by drilling in multi-dimensional space to get deep insights for business analysis (Ex. 2).

Example 1. (Multi-dimensional skyline query) Consider a used car database (*e.g.*, kbb.com) with schema (*type, maker, color, price, milage*). The first three are boolean dimensions, whereas the last two are preference ones. A static skyline query may ask for skyline (on *price* and *milage*) objects among a subset of cars with *type = "sedan" and color = "red"*. A dynamic skyline query has more flexibility in expressing user preference:

select dynamic skylines from R
where type = "sedan" and color = "red"
preference by $(price - 10k)^2$ and $(milage - 10k)^2$

This query asks for every red sedan whose *price* is close to $10k$ and whose *milage* is close to $10k$, and there is no other red sedan which is closer on both preferences. ■

Example 2. (Multi-dimensional skyline comparison) Consider a digital camera comparison database (*e.g.*, bizrate.com) with schema (*brand, type, price, resolution, optical zoom*). Suppose the last three dimensions are preference dimensions. An analyzer who is interested in canon professional cameras may first issue a skyline query with boolean predicate *type = "professional" and brand = "canon"*. The analyzer then rolls up on the *brand* dimension and checks the skylines of professional cameras by all makers. By comparing two sets of skylines, the analyzer will find out the position of canon cameras in the professional market.

■

As shown in the above examples, the static skylines are fixed for a given data-set. The dynamic skylines support ad-hoc user preferences and are more powerful in applications.

7.1.3 Query model

Consider a relation R with *boolean dimensions* $A_1, A_2, \ldots, A_b$, and *preference dimensions* (*i.e.*, ranking dimensions in top-k query) $N_1, N_2, \ldots, N_p$. The two sets of dimensions are not necessarily exclusive. A dynamic skyline query specifies the boolean predicates on a subset of boolean dimensions and preference functions on a subset of preference dimensions. A possible SQL-like notation for expressing dynamic skyline queries is as follows:

select dynamic skylines from R
where $A'_1 = a_1$ and $\cdots$ and $A'_i = a_i$
preference by $f_1, f_2, \ldots, f_m$

where $\{A'_1, A'_2, \ldots, A'_i\} \subseteq \{A_1, A_2, \ldots, A_b\}$ and $f_1, f_2, \ldots, f_m$ are formulated on $\{N_1, N_2, \ldots, N_p\}$. Without losing generality, we assume that users prefer *minimal* scores. The query results are a set of objects that belong to the data set satisfying the boolean predicates and are not dominated by any other objects in the same set.

We assume that functions $f_1, f_2, \ldots, f_m$ have the following property: Given a function $f(N'_1, N'_2, \ldots, N'_i)$ and the domain region Ω on its variables, the lower bound of f over Ω can be derived. For many continuous functions, this can achieved by computing the derivatives of f.

To support such on line analytical domination queries, it is important to restructure the source data such that both multi-dimensional boolean predicates and dynamic skyline queries can be addressed simultaneously. In this chapter, we adopt the ranking-cube for answering preference queries. In the rest of this chapter, we first presents the method for static skyline queries, and then discuss the extension to dynamic skylines. We report the experimental results in Section 7.3.

7.2 OLAPing Skyline Queries

7.2.1 Problem Analysis

Given the boolean and domination criteria, an algorithm can first filter data tuples by boolean predicates and then compute the skylines (*i.e.*, boolean pruning first). Alternatively, one can search data tuples according to the domination relationship and verify the boolean constraints on each candidate (*i.e.*, domination pruning first). This is similar to our analysis in top-k query processing.

We use the signature-based ranking-cube (Chapter 4) in this chapter, and adopt the branch-and-bound search paradigm. Based on the R-tree partition, [51] proposed a branch-and-bound search paradigm which starts from the root node of the R-tree and progressively expands a node by examining its child nodes. A node is pruned if it is dominated by some other data points. The search halts when there is no node left. To integrate boolean pruning in the above search framework, the algorithm can use signature to identify whether an underlying node contains any object satisfying the boolean predicates.

7.2.2 Querying Static Skylines

We outline the *signature-based* query processing in Algorithm 7. The algorithm follows the branch-and-bound principle to progressively retrieve data nodes. To demonstrate the correctness of the algorithm, we first present two facts. First, if a node n is dominated by some data objects, all child nodes of n are dominated. Secondly, for each node n, let $d(n) = \min_{x \in n}(\sum_{i=1}^{m} f_i^2(x))$ be the lower bound value over the region covered by n. A data object t can not be dominated [51] by any data objects contained by node n if $\sum_{i=1}^{m} f_i^2(t) \leq d(n)$.

We briefly explain each step as follows. Line 1 initializes a list to store the final results. Line 2 loads signature measure from ranking-cube according to the boolean predicates BP. The construction and retrieval of signature measure is presented in Chapter 4. Each node n

Algorithm 7 Framework for Query Processing

Input: R-tree R, ranking-cube C, boolean predicates BP, and
a set of preference functions F

```
1: sky = φ; // initialize the result set
2: sig = signature measure of BP in C; // load signature
3: c_heap = {R.root}; // initialize candidate heap
4: while (c_heap ≠ φ)
5:    remove top entry e;
6:    if (prune(e, sky, sig))
7:      continue;
8:    if (e is a data object)
9:      insert e into sky;
10:   else // e is a node
11:     for each child e_i of e // expand the node
12:       if (!prune(e_i, sky, sig))
13:         insert e_i into c_heap;
14: return
```

```
Procedure prune(e, sky, sig)
  Global Lists: b_list, d_list;
15: if (e is dominated by sky) // domination checking
16:    insert e into d_list;
17:    return true;
18: if (e does not satisfy boolean predicates) // check sig
19:    insert e into b_list;
20:    return true;
21: return false;
```

is associated with a value $d(n)$, and the root of the *c_heap* contains an entry e with minimal $d(e)$ (line 5). Line 6 checks whether all tuples contained by e satisfy BP, or are dominated by previously retrieved data objects (as we mentioned above, e can not be dominated by any future data objects). For all e that pass the checking, e is a new skyline object if it is a tuple (line 8). Otherwise, the algorithm further examine e's child nodes (lines 11-13).

The *prune* procedure is the key component in the framework. An algorithm is *optimal* if *prune* procedure does not return false positives. The signature measure provides correct answers for boolean pruning. While the domination pruning for static skylines is straightfor-

ward, it is more complicated for dynamic skylines. In Section 7.2.3, we explain why this is the case and discuss how to reduce the false positives. The *prune* procedure also maintains two optional global lists: the *b_list* and *d_list*. The *b_list* keeps all the entries pruned by boolean predicates and the *d_list* keeps all the entries pruned by skyline domination. The sole purpose of maintaining these two lists is to efficiently support roll-up and drill-down queries (Section 7.2.4).

7.2.3 Querying Dynamic Skylines

Here we discuss the domination pruning for dynamic skylines. The difficulty lies on the fact that an entry e submitted to the *prune* procedure may be an intermediate node, which corresponds to a region. In static skyline computation, this region is a rectangle. While in dynamic skyline computation, the region is mapped by preference functions and could form arbitrary shape.

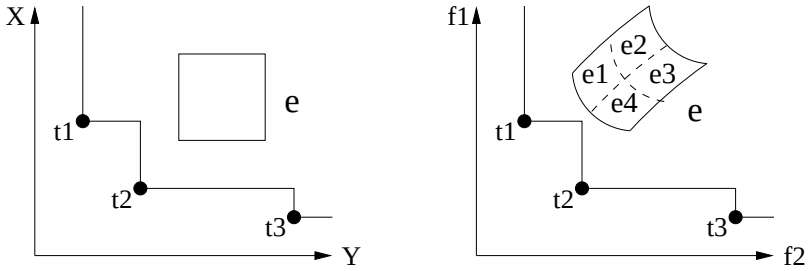


Figure 7.1: **Domination Pruning**

Suppose the preference functions for a dynamic skyline query are $f_1 = X^2 + Y^2$ and $f_2 = (10 - X)^2 + Y^2$, where X and Y are two preference dimensions. Figure 7.1 shows the difference of pruning between static skylines and dynamic skylines. Suppose t_1, t_2 and t_3 are skyline objects, and e is an entry submitted for domination checking. In static skyline checking, if e does not contain any possible skyline result, there must exist one existing skyline object that *completely* dominates e. Consequently, the *prune* procedure does not have false positive. In dynamic skyline checking, though the entry e should be pruned, there

is no single existing skyline object that completely dominates e. Instead, t_1 dominates a sub-region of e and t_2 dominates the rest.

To analytically determine whether a region with arbitrary shape is dominated by a set of skyline objects is a challenge task. Instead, we present an alternative sub-optimal solution. Motivated by the fact that the whole region is dominated by multiple objects, and each of which dominates a sub-region, we evenly partition e into sub-regions e_1, e_2, e_3 and e_4 (Figure 7.1). We observe that e_1 is dominated by t_1, e_4 is dominated by t_2, e_2 and e_3 are dominated by both t_1 and t_2. Thus, we can safely prune e. In the case when there is a sub-region not being dominated, we can further partition it. The algorithm specifies a threshold k to control the depth of the recursive partition.

7.2.4 Drill Down and Roll up Queries

Drill-down and roll-up are typical OLAP operators applied on boolean dimensions. Given the current boolean predicate BP, the drill-down query strengthens BP by augmenting an additional boolean predicate and roll-up query relaxes BP by removing some boolean predicate in BP. For example, let $BP = \{A = a_1, B = b_1\}$. $BP' = \{A = a_1, B = b_1, C = c_1\}$ is a drill-down, whereas $BP' = \{A = a_1\}$ is a roll-up. Drill-down and roll-up queries are incremental in that they always follow a standard skyline query. Moreover, for both queries, we assume that the preference functions for dynamic skylines keep the same.

A standard skyline query starts with an empty *c_heap* and searches from the root node. While in drill-down and roll-up queries, we can avoid searching from scratch. Thus, the I/O cost may be reduced. Recall that in Algorithm 7, we maintain three lists: *sky*, *b_list* and *d_list*, where *sky* contains the results for current query, *d_list* contains entries dominated by objects in *sky*, and *b_list* contains entries not satisfying boolean predicates. Lemma 9 shows how to re-construct *c_heap* without starting from the root node.

Lemma 9 *Suppose sky, b_list and d_list are maintained by the last query. For a continuing*

drill-down (or roll-up) query, re-constructing the candidate heap as c_heap = sky ∪ d_list (or c_heap = sky ∪ b_list) returns the correct answers.

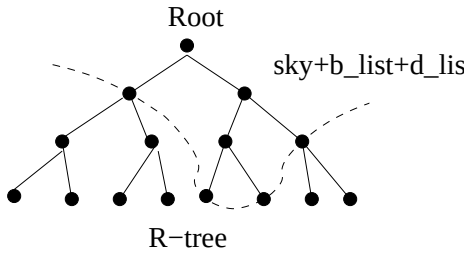


Figure 7.2: **Re-constructing candidate heap**

proof We prove the drill-down case, and the proof for roll-up is similar. Let the union of *sky*, *b_list* and *d_list* be N. According to Algorithm 7, each retrieved node is either expanded (by inserting child nodes to *c_heap*) or inserted into N. When the search completes, *c_heap* is empty. Thus N forms a closure (*e.g.*, Figure 7.2) such that for any unvisited node n, there is a node $n' \in N$ and n' is an ancestor of n. If we re-construct *c_heap* by N, we will not miss any unvisited node and all skyline objects will be discovered. Since drill-down query strengthens the boolean condition, all entries in *b_list* (and their descendants) are pruned by the new boolean predicates. Thus we can remove *b_list* from *c_heap*. ■

With the re-constructed *c_heap*, we execute Algorithm 7 from Line 4. Note the size of *c_heap* can be further reduced by enforcing boolean checking and domination checking beforehand. Using drill-down as example, for each previous skyline object in *sky*, it continues to be a skyline objects if it satisfies the drill-down boolean predicate. Otherwise, it is directly moved to *b_list*. For each entry in *d_list*, we filter it by the *prune* procedure before we insert it into *c_heap*.

7.3 Performance Study

This section reports our experimental results. We compare the query performance of Algorithm 7 with two other alternatives: (1) the *boolean-first* approach that evaluates boolean

predicates before the skyline computation, and (2) the *domination-first* approach that conducts boolean verification after each candidate skyline is generated. We first discuss the experimental settings, and then show the computation and space costs of ranking-cube, and online query performance.

7.3.1 Experimental Setting

We defines the data sets and the experimental configurations for all approaches.

Data Sets

We use both synthetic and real data sets for the experiments. The real data set we consider is the *Forest CoverType* data set obtained from the UCI machine learning repository web-site (www.ics.uci.edu/∼mlearn). This data set contains $581,012$ data points with 54 attributes. We select 3 quantitative attributes (with cardinalities $1,989$, $5,787$ and $5,827$) as preference dimensions, and other 12 attributes (with cardinalities 255, 207, 185, 67, 7, 2, 2, 2, 2, 2, 2, 2) as boolean dimensions. We also generate a number of synthetic data sets for our experiments. For each synthetic data, $\mathcal{D}_p$ denotes the number of preference dimensions, $\mathcal{D}_b$ the number of boolean dimensions, $\mathcal{C}$ the cardinality of each boolean dimension, $\mathcal{T}$ the number of tuples, $\mathcal{S} = \{E, C, A\}$ the uniform, correlated and anti-correlated data distributions.

Experimental Configurations

We build all *atomic* cuboids (*i.e.*, all single dimensional cuboids on boolean dimensions) for ranking-cube. Signatures are compressed, decomposed and indexed (using B_+-tree) by cell values and SID's. The page size in R-tree is set as $4KB$. In the experiments, we compare our proposed signature-based approach (referred as *Signature*) against the boolean-first (referred as *Boolean*) and domination-first (referred as *Domination*) approaches.

Boolean first: We use B_+-tree to index each boolean dimension. Given the boolean predicates in query, we first select tuples satisfying the boolean conditions. To process

boolean filters, we may use index scan or table scan, we report the best performance of the two alternatives. The selected data tuples are inserted into a candidate heap (*i.e.*, *c_heap*). The key value for comparison in the heap is $d(t) = \sum_{i=1}^{m} f_i^2(t)$. We fetch and remove tuples from *c_heap* one by one, and compute the skyline objects. In this approach, we assume *c_heap* fits in memory. A simple optimization is applied to improve the performance: let $d_{max}(t) = \max_{i=1}^{m} f_i(t)$ and $d_{min}(t) = \min_{i=1}^{m} f_i(t)$. During tuple retrieval, we maintain the minimal value (*i.e.*, *minmax*) of d_{max} among all seen tuples. For each arrival tuple t, if $d_{min}(t) > minmax$, t can be directly discarded without inserted into *c_heap* since it must be dominated by some other tuples.

Domination first: We adopted the bbs algorithm [51], which is best known in the literature. The algorithm progressively retrieves R-tree blocks until the results are computed. The framework is similar to Algorithm 7, except that there is no boolean checking in the *prune* procedure. Since R-tree only keeps values in preference dimensions, we build index on *tid* for the database and keep *tid* with each tuple in R-tree for boolean verification. The boolean verification involves random access and we only issue a boolean checking for a tuple in between line 8 and line 9 in Algorithm 7. That is, we only verify a tuple which has been determined as a candidate skyline. In this way, one can prove that the number of boolean verification is *minimized*. One may suggest to keep boolean dimensions in R-tree for cheap verification. This approach may not be a sound solution in reality: first, it reduces the capacity of each node, and the size of R-tree may increase a lot; and secondly, it may violate some constraints (*e.g.*, the tuples must be sorted by a primary key) and introduce difficulties for other types of data access (*e.g.*, sequential table scan).

7.3.2 Experimental Results

Experiments are conducted to examine (1) the query performance on static skylines, (2) the query performance on dynamic skylines, and (3) the effect of boolean predicates on query performance, including the drill-down and roll-up queries.

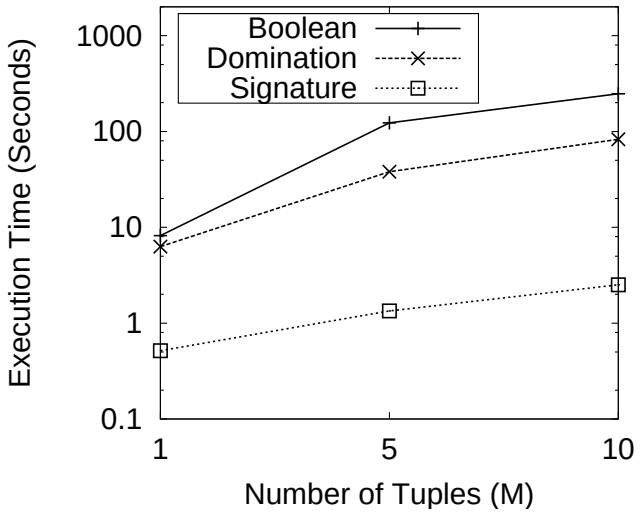


Figure 7.3: **Execution Time w.r.t. $\mathcal{T}$**

Query Performance on Static Skylines

To begin with, we examine the query performance on static skylines. All queries use single selection condition on one boolean dimension. The results on multiple boolean predicates are reported in Section 7.3.2.

We run static skyline queries on the same synthetic data sets as those in Figure 4.8, and the execution time is shown in Figure 7.3. We compare *Signature* with *Boolean* and *Domination*. Clearly, the signature-based query processing is at least one order of magnitude faster. This is because in *Boolean*, disk access is based on boolean predicates only, and in *Domination*, disk access solely relies on domination analysis. *Signature* combines both pruning opportunities and thus avoids unnecessary disk accesses.

To take a closer look, we compare the number of disk accesses between *Signature* and *Domination* in Figure 7.4. *Domination* consists of two types of disk accesses: R-tree block retrieval (*DBlock*) and random tuple access for boolean verification (*DBool*). *Signature* also consists of two types of disk access: signature loading (*SSig*) and R-tree block retrieval (*SBlock*). We observe that (1) in *Signature*, the cost of loading signature is far smaller ($\leq 1\%$) than that of retrieving R-tree blocks, and (2) guided by the signatures, our method

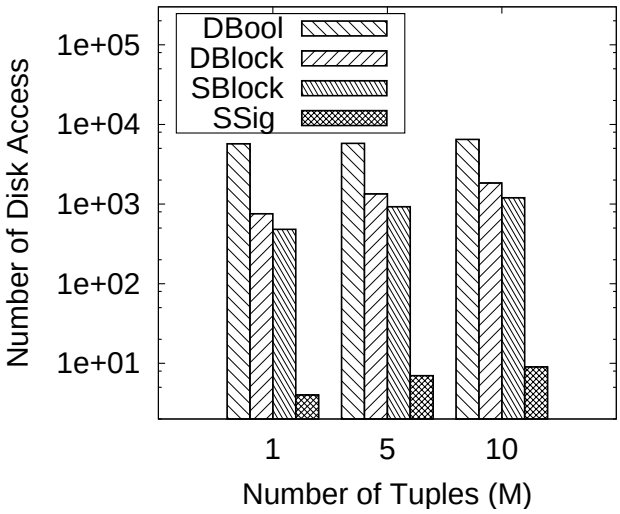


Figure 7.4: **Number of Disk Access w.r.t. $\mathcal{T}$**

prunes more than 1/3 R-tree blocks comparing with *Domination* and avoids even more random tuple accesses for boolean verification.

From another perspective, reducing the memory requirement is equally important for the scalability issue. Note this is not necessarily implied by minimizing disk accesses. For example, we adopt a lazy verification strategy in *Domination* and this has trade-off of keeping more candidates in heap. Figure 7.5 compares the peak size of candidate heap in memory for all three methods. With *Signature*, the number of entries kept in memory is an order of magnitude less than that of *Domination* and *Boolean*.

The performance of *Boolean* and *Domination* depends on the boolean and preference selectivity, respectively. The boolean (preference) selectivity determines the filtering power by boolean predicates (multi-dimensional dominations). We first vary the cardinality $\mathcal{C}$ of each boolean dimension from 10 to 1000, while keeping $\mathcal{T} = 1M$. The query execution time is shown in Figure 7.6. As expected, *Boolean* performs better when $\mathcal{C}$ increases and the performance of *Domination* deteriorates. The preference selectivity is affected by the data distribution among preference dimensions and also the number of preference dimensions. We generate two sets of synthetic data: (1) with correlated, uniform and anti-correlated distri-

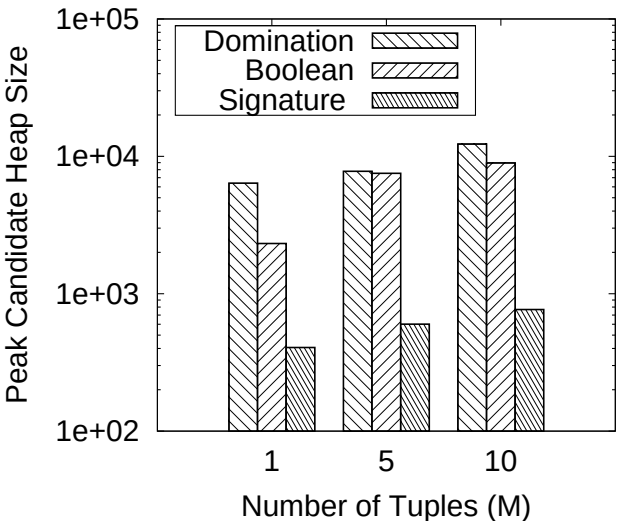


Figure 7.5: **Peak Candidate Heap Size w.r.t.** $\mathcal{T}$

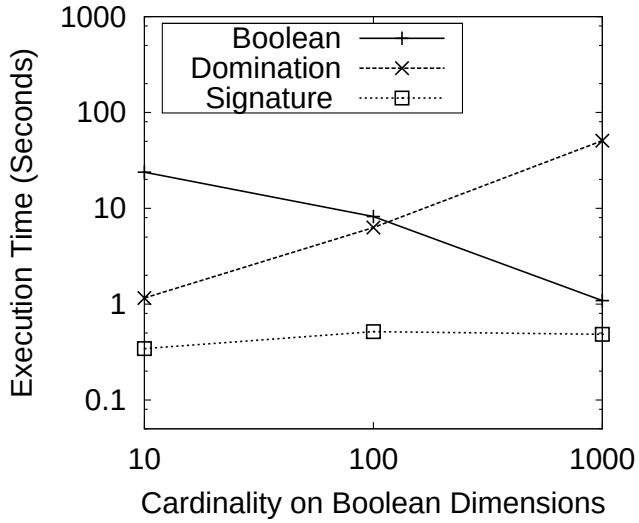


Figure 7.6: **Execution Time w.r.t.** $\mathcal{C}$

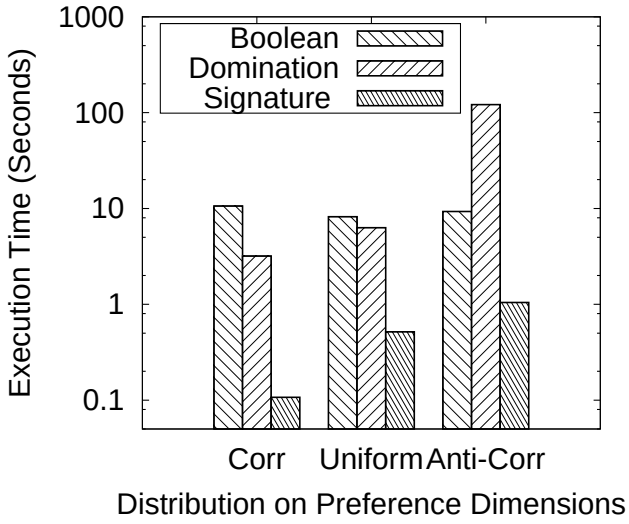


Figure 7.7: **Execution Time w.r.t.** $\mathcal{S}$

butions on preference dimensions (by fixing $\mathcal{D}_p = 3$), and (2) with the number of preference dimension varying from 2 to 4 (by fixing uniform distribution). The query performance is shown in Figures 7.7 and 7.8, respectively. When data is correlated or the number of preference dimensions is low, it is easier to find static skylines. When data is anti-correlated or the number of preference dimensions increases, domination relationship between objects is weaker. As the result, it becomes more challenging to compute the skyline results, and the computation time for *Domination* increases. On the other hand, the preference selectivity has limited effect on *Boolean*. In all experiments, *Signature* performs fairly robustly and is consistently the best among the three.

Query Performance on Dynamic Skylines

We continue to evaluate the query performance on dynamic skylines. The algorithms have similar behaviors as those in static skylines in many tests, such as scalability and boolean selectivity. The preference selectivity can be affected by preference functions specified by a user, since they may map the data to an arbitrary space.

To demonstrate this, we use distance functions for preference functions in our experi-

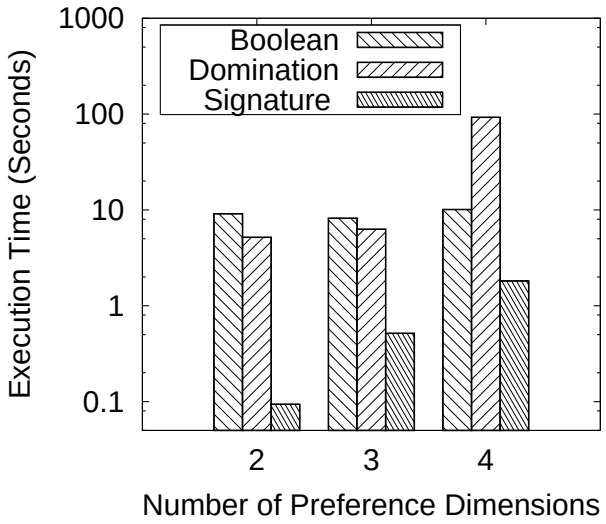


Figure 7.8: **Execution Time w.r.t.** $\mathcal{D}_p$

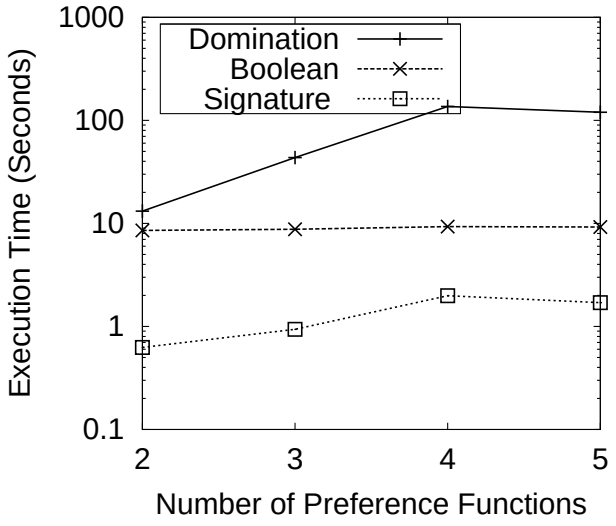


Figure 7.9: **Execution Time w.r.t.** m

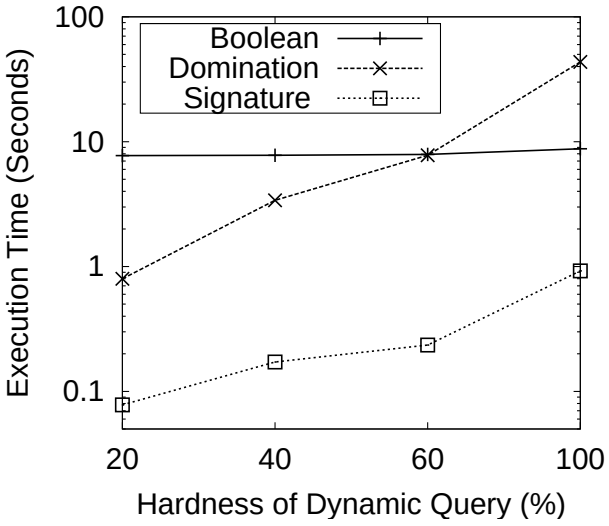


Figure 7.10: **Execution Time w.r.t. Hardness**

ments. Given a data set with $\mathcal{D}_p$ preference dimensions, a distance function f_i measures the distance between a tuple t and a fixed $\mathcal{D}_p$ dimensional point P_i by $f_i(t) = \sum_{j=1}^{\mathcal{D}_p}(t^j - P_i^j)^2$, where t^j and P_i^j are the values on the j^{th} preference dimension of the tuple t and data point P_i. We first vary the number of preference functions m from 2 to 5. The synthetic data set has $1M$ tuples, 3 preference dimensions, 3 boolean dimensions (with cardinality 100). The data point P_i for each f_i is randomly generated. The query execution time is shown in Figure 7.9. The curve trends are similar to those shown in Figure 7.8, and the signature-based approach again performs the best.

The above experiment also suggests an important application scenario for dynamic skyline. A well known short-coming of static skyline is the curse of dimensionality: There are too many skyline objects in high-dimensional data since objects are difficult to dominate each other. Dynamic skylines are more attractive in high-dimensional space since one can combine several dimensions together by a single preference function. The number of dimensions in the mapped space is reduced, and so is the number of skyline objects. For example, in our experiment, the number of skyline objects decreases from 602 ($m = 5$) to 95 ($m = 2$). In case where only one preference function is used, the query is identical to a top-1 ranked

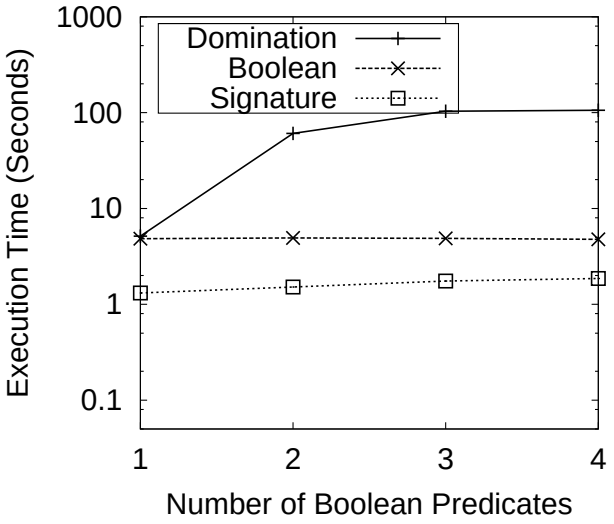


Figure 7.11: **Execution Time w.r.t. Boolean Predicates**

query.

We can also simulate different data distributions on the mapped space by varying the distance between P_i's in preference functions. When P_i's are closer to each other, the mapped data forms a more correlated distribution since any tuple t gets closer values on all f_i. We refer this as query hardness and measure it by maximal distance[1] between P_i's. Fixing the number of preference functions as 3, we vary the maximal distance allowed between P_i's. The query performance on the same data set is shown in Figure 7.10. As expected, the curves are similar to those in Figure 7.7.

Query Performance with Boolean Predicates

The last group of experiments evaluates the query performance w.r.t. boolean predicates. We use the real data set *Forest CoverType*, which consists of 12 boolean dimensions and 3 preference dimensions.

We issue dynamic skyline queries with 1 to 4 boolean predicates and the execution time is shown in Figure 7.11. *Signature* and *Boolean* are not sensitive to boolean predicates, and the former performs consistently better. *Domination* requests more boolean verification, and

[1]The domain of maximal distance is scaled to $[0, 1]$

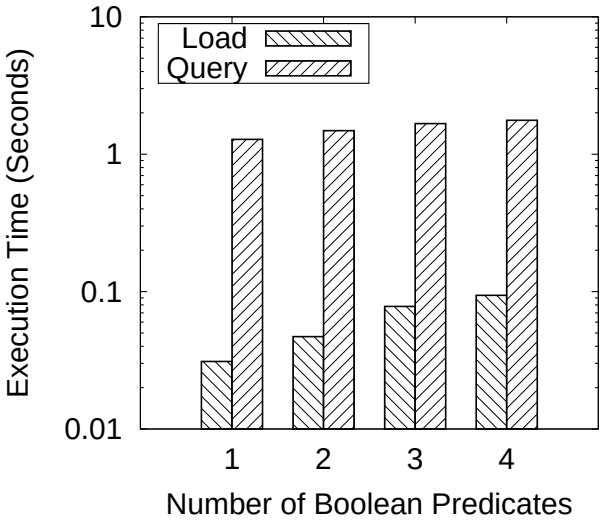


Figure 7.12: **Signature Loading Time vs. Query Time**

thus the execution time grows significantly. When there are $k > 1$ boolean predicates, we essentially need to load k one-dimensional signatures since only atomic cuboids are materialized. The comparison of execution time used by signature loading and query processing is shown in Figure 7.12. The time used for loading signatures increases slightly with k. However, even when there are 4 boolean predicates, the signature loading time is still far less than the query processing time (*i.e.*, less than 10%). Figure 7.12 suggests that materializing atomic cuboids only may be good enough in real applications. This also reduces the costs for off-line computation and incremental maintenance.

Figures 7.13 and 7.14 show the performance gains of drill-down and roll-up queries over new queries, respectively. For each query with k ($k \geq 2$) boolean predicates in Figure 7.11, the drill-down query is executed in two steps: (1) submit a query with $k-1$ boolean predicates, and (2) submit a drill-down query with the additional k^{th} boolean predicate. Similarly, for each query with k ($k \leq 3$) boolean predicates, the roll-up query is executed by first submitting a query with $k+1$ boolean predicates, and then rolling-up on the $(k+1)^{th}$ boolean dimension. In most cases, we observe more than 10 times speed-up by caching the previous intermediate results and re-constructing candidate heap upon them. In general,

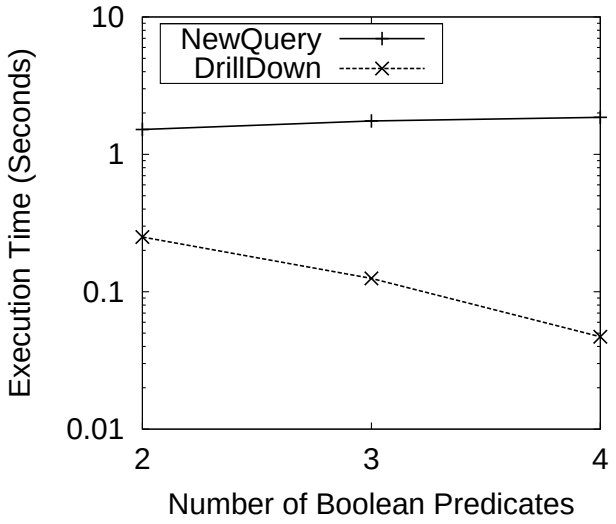


Figure 7.13: **Drill-Down Query vs. New Query**

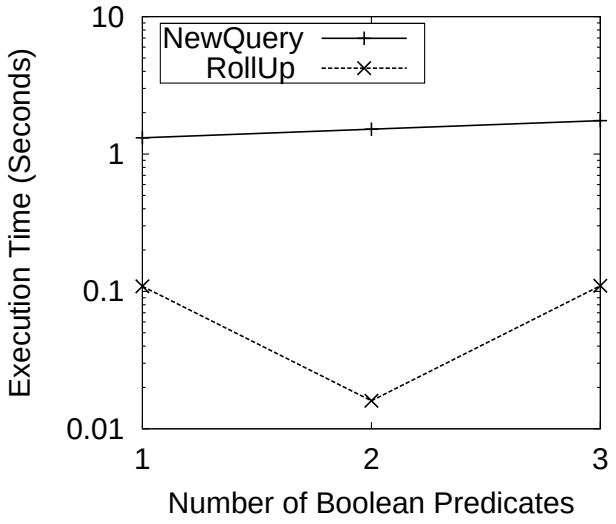


Figure 7.14: **Roll-Up Query vs. New Query**

the margin of speed-up depends on certain data properties such as cardinalities and data correlation.

7.4 Discussion

7.4.1 Related Work

There have been extensive studies on efficiently computing skylines in database systems, including divide-and-conquer and block nested loop algorithms by Borzsonyi *et al.* [12], sort-first skyline by Chomicki *et al.* [26], bitmap and index methods by Tan *et al.* [62], nearest neighbor approach by Kossmann *et al.* [42] and branch-and-bound search by Papadias *et al.* [51]. All these approaches assume the queries consist preference dimensions only, and the problem of skyline queries with boolean predicates was not well addressed. For example, the branch-and-bound search algorithm progressively retrieves R-tree blocks until all the skylines are found. The method is optimal in that it only accesses blocks that may contain skyline objects. The definition of dynamic skyline was first proposed in [51] which shows that the same branch-and-bound approach can be used for dynamic skylines queries. Unfortunately, as shown later in this chapter, the optimality of the algorithm does not hold if the query involves either boolean predicates or dynamic skylines. The former is true because it may retrieve blocks which contain no tuple satisfying boolean conditions; and the latter is true because of the new challenge of domination checking introduced by dynamic mapping.

Skyline computation is also studied under the multi-dimensional context. Tao *et al.* [63] studied subspace skyline computation, Yuan *et al.* [74] exploited computation sharing for skyline cube, Chan *et al.* [19] addressed k-dominated skylines in high dimensional data, and Pei *et al.* [52] studied the semantics of skyline cube. A data cube design for domination analysis queries was proposed by Li *et al.* [46]. Again, all these studies are conducted under the context that there is no boolean predicate.

The dynamic skyline query builds a bridge between traditional static skyline and top-k

queries. In skyline query, one is interested in multi-dimensional domination, while in top-k query, one is interested in a combined ranking criterion over several participating dimensions. A top-1 query is essentially a special dynamic skyline query with only one preference function. To extend 1 to k, we only need to make a slight modification on the *prune* procedure in Algorithm 7. For the domination checking, we will keep a list of the current top-k results and prune an entry if its score is worse than the k^{th} score in the list.

7.4.2 Other Preference Queries

Here we briefly discuss how to extend Algorithm 7 to support other preference queries, such as skyline queries [51] and convex hull queries [11]. We briefly use convex hull query as an example.

In [11], Bohm and Kriegel proposed an algorithm for progressively computing convex hulls in databases indexed by R-trees. The algorithm decomposes the complete convex hull into $2^{\mathcal{D}_p}$ partial convex hulls by the extreme values on each preference dimension. We can adopt Algorithm 7 to compute each partial convex. Again, the signature-based boolean checking can be used without any modification. For the domination checking, one can keep a list of the current objects which constitute a convex hull. A new entry is pruned if it is entirely inside the hull.

Chapter 8

Conclusions and Future Work

To efficient process top-k queries with multi-dimensional selections, we proposed a novel rank-aware cube structure which is capable to simultaneously handle ranked queries and multi-dimensional selections. The ranking-cube consists of two components: the data partition and the cube measure that summarizes data distribution with respect to multi-dimensional group-bys. We motivated our research by a simple grid-partition, and used ID-list as measures. We then extended the framework by using hierarchical partition, and generating compact signature as measures.

Based on the ranking cube, we developed efficient ranked query processing algorithms. For grid-partition, we adopted the neighborhood search, and for hierarchical partition, we adopted the branch-and-bound search. Both search algorithms are optimal in terms of the number of partition blocks retrieved from the disk. Our experimental results show that the proposed methods significantly improve the query performance over the previous approaches.

We discussed the extensions of ranking-cube to high dimensional data. For high selection dimensions, we extended the ranking cube to ranking fragments, where many ranking fragments share a single data partition. for high ranking dimension, we extended the sort-merge paradigm to index-merge framework. We addressed two challenges within the index-merge paradigm. First, to reduce the search complexity, we developed a double heap algorithm that consists of the neighborhood expansion and the threshold expansion. Second, to avoid retrieving empty-state, we proposed join-signature which is compact, easy to compute and incurs low overhead in query processing.

Finally, we extended the ranking cube methodology to processing top-k queries in *multiple*

relations by bridging multiple ranking cubes, each of which is derived from its corresponding relation. Based on this new methodology, we build a *SPJR* (namely, Selection, Projection, Join and Ranked queries) query evaluation system which consists of a *query optimizer* and a *query executer*. We also demonstrated how to use the ranking-cube model to answer general preference queries.

There are many interesting research issues on further extensions of the ranking-cube methodology. It will be useful to integrate our method to current DBMS. Ranking-cube can be either implemented as an external DB application or as part of the core database engine. Currently, we implemented ranking-cube in two ways. The grid partition based version is implemented as an external DB application, where each cuboid is a relational table. We use SQL operators to access ranking cube and the data. The hierarchical partition based version is implemented directly on file systems, and we use R-tree and B-trees to index the ranking cube and data. It is interesting to integrate ranking-cube to the core DBMS engine. It is also interesting to identify cases where the ranking-cube is not beneficial. By doing this, the query optimize may choose the right plan in query processing.

It is also interesting and important to exploit ranking-cube method on IR-style applications, typically on text data. Take news data as example. One can rank documents by there relevance to the query terms, and one can also impose hard selection conditions such as document type, time, source, *etc.*. In text retrieval, the ranking criteria are mostly the relevance to the query, and the ranking function may belong to certain classes. The system can leverage this and design more effective data partitioning strategy.

References

[1] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. 1996 Int. Conf. Very Large Data Bases (VLDB'96)*, pages 506–521, Bombay, India, Sept. 1996.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases (VLDB'94)*, pages 487–499, Santiago, Chile, Sept. 1994.

[3] S. Amer-Yahia and T. Johnson. Optimizing queries on compressed bitmaps. *Proceedings of 2000 International Conference on Very Large Data Bases (VLDB'00)*, pages 329–338, 2000.

[4] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multidimensional database. In *Proc. 1997 Int. Conf. Very Large Data Bases (VLDB'97)*, pages 98–12, Athens, Greece, Aug. 1997.

[5] D. Barbara and M. Sullivan. Quasi-cubes: Exploiting approximation in multidimensional databases. *SIGMOD Record*, 26:12–17, 1997.

[6] D. Barbará and X. Wu. Using loglinear models to compress datacube. In *Proc. 1st Int. Conf. Web-Age Information Management (WAIM'2000)*, pages 311–322, Shanghai, China, 2000.

[7] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *Proc. 2006 Int. Conf. Very Large Data Bases (VLDB'06)*, pages 475–486, 2006.

[8] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. 1990 ACM SIGMOD Int. Conf. Management of Data (SIGMOD'90)*, pages 322–331, Atlantic City, NJ, June 1990.

[9] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. 1999 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'99)*, pages 359–370, 1999.

[10] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[11] C. Bohm and H.-P. Kriegel. Determining the convex hull in large multidimensional databases. In *Proceedings of 2000 International Conference on Data Warehousing and Knowledge Discovery (DaWaK'00)*, pages 294–306. Springer-Verlag, 2001.

[12] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. pages 421–430, 2001.

[13] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. pages 263–274, 2002.

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

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

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

[17] K. Chakrabarti, V. Ganti, J. Han, and D. Xin. Ranking objects based on relationships. In *Proc. 2006 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'06)*, pages 371–382, 2006.

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

[19] C.-Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. Finding k-dominant skylines in high dimensional space. In *Proc. 2006 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'06)*, pages 503–514. ACM Press, 2006.

[20] K. C.-C. Chang and S. won Hwang. Minimal probing: Supporting expensive predicates for top-k queries. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*, pages 346–357, 2002.

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

[22] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. *Proceedings of 1999 International Conference on Very Large Data Bases (VLDB'99)*, pages 397–410, 1999.

[23] B.-C. Chen, L. Chen, Y. Lin, and R. Ramakrishnan. Prediction cubes. *Proceedings of 2005 International Conference on Very Large Data Bases (VLDB'05)*, pages 982–993, 2005.

[24] Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang. Multi-dimensional regression analysis of time-series data streams. In *Proc. 2002 Int. Conf. Very Large Data Bases (VLDB'02)*, pages 323–334, Hong Kong, China, Aug. 2002.

[25] Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang. Multi-dimensional regression analysis of time-series data streams. *Proceedings of 2002 International Conference on Very Large Data Bases (VLDB'02)*, pages 323–334, 2002.

[26] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *Proc. 2003 Int. Conf. Data Engineering (ICDE'03)*, pages 717–816, 2003.

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

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

[29] R. Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31(2):109–118, 2002.

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

[31] A. Fraenkel and S. Klein. Novel compression of sparse bit-strings - preliminary report. *Combinatorial Algorithms on Words, NATO ASI Series*, 12:169–183, 1985.

[32] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[33] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-totals. *Data Mining and Knowledge Discovery*, 1:29–54, 1997.

[34] H. Gupta. Selection of views to materialize in a data warehouse. In *Proc. 7th Int. Conf. Database Theory (ICDT'97)*, pages 98–112, Delphi, Greece, Jan. 1997.

[35] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *Proc. 1997 Int. Conf. Data Engineering (ICDE'97)*, pages 208–219, Birmingham, England, April 1997.

[36] A. Guttman. R-tree: A dynamic index structure for spatial searching. In *Proc. 1984 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'84)*, pages 47–57, Boston, MA, June 1984.

[37] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. In *Proc. 2001 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'01)*, pages 1–12, Santa Barbara, CA, May 2001.

[38] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'96)*, pages 205–216, Montreal, Canada, June 1996.

[39] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'98)*, pages 237–248, 1998.

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

[41] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid. Rank-aware query optimization. *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 203–214, 2004.

[42] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proc. 2002 Int. Conf. Very Large Data Bases (VLDB'02)*, pages 275–286, 2002.

[43] L. V. S. Lakshmanan, J. Pei, and J. Han. Quotient cube: How to summarize the semantics of a data cube. In *Proc. 2002 Int. Conf. on Very Large Data Bases (VLDB'02)*, pages 778–789, Hong Kong, China, Aug. 2002.

[44] L. V. S. Lakshmanan, J. Pei, and Y. Zhao. QC-Trees: An efficient summary structure for semantic OLAP. In *Proc. 2003 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'03)*, pages 64–75, San Diego, CA, June 2003.

[45] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. Ranksql: Query algebra and optimization for relational top-k queries. *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD'05)*, pages 131–142, 2005.

[46] C. Li, B. C. Ooi, A. K. H. Tung, and S. Wang. Dada: a data cube for dominant relationship analysis. In *Proc. 2006 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'06)*, pages 659–670, 2006.

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

[48] K. Morfonios and Y. Ioannidis. Cure for cubes: cubing using a rolap engine. In *Proc. 2006 Int. Conf. Very Large Data Bases (VLDB'06)*, pages 379–390, 2006.

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

[50] R. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'98)*, pages 13–24, Seattle, WA, June 1998.

[51] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

[52] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *Proc. 2005 Int. Conf. Very Large Data Bases (VLDB'05)*, pages 253–264, 2005.

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

[54] K. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proc. 1997 Int. Conf. Very Large Data Bases (VLDB'97)*, pages 116–125, Athens, Greece, Aug. 1997.

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

[56] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*, pages 23–34, 1979.

[57] J. Shanmugasundaram, U. M. Fayyad, and P. S. Bradley. Compressed data cubes for OLAP aggregate query approximation on continuous dimensions. In *Proc. 1999 Int. Conf. Knowledge Discovery and Data Mining (KDD'99)*, pages 223–232, San Diego, CA, Aug. 1999.

[58] H. Shin, B. Moon, and S. Lee. Adaptive and incremental processing for distance join queries. *IEEE Trans. Knowl. Data Eng.*, 15(6):1561–1578, 2003.

[59] A. Shukla, P. M. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 488–499, New York, NY, Aug. 1998.

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

[61] Y. Sismanis, N. Roussopoulos, A. Deligianannakis, and Y. Kotidis. Dwarf: Shrinking the petacube. In *Proc. 2002 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'02)*, pages 464–475, Madison, WI, June 2002.

[62] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *Proc. 2001 Int. Conf. Very Large Data Bases (VLDB'01)*, pages 301–310, 2001.

[63] Y. Tao, X. Xiao, and J. Pei. Subsky: Efficient computation of skylines in subspaces. In *Proc. 2006 Int. Conf. Data Engineering (ICDE'06)*, page 65, 2006.

[64] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava. Ranked join indices. pages 277–288, 2003.

[65] P. Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, 1987.

[66] J. S. Vitter, M. Wang, and B. R. Iyer. Data cube approximation and histograms via wavelets. In *Proc. 1998 Int. Conf. Information and Knowledge Management (CIKM'98)*, pages 96–104, Washington, DC, Nov. 1998.

[67] W. Wang, H. Lu, J. Feng, and J. X. Yu. Condensed cube: An effective approach to reducing data cube size. In *Proc. 2002 Int. Conf. Data Engineering (ICDE'02)*, pages 155–165, San Fransisco, CA, April 2002.

[68] D. Xin, C. Chen, and J. Han. Towards robust indexing for ranked queries. In *Proc. 2006 Int. Conf. Very Large Data Bases (VLDB'06)*, pages 235–246, 2006.

[69] D. Xin, J. Han, and K. C.-C. Chang. Progressive and selective merge: Computing top-k with ad-hoc ranking functions . In *Proc. 2007 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'07)*, pages 103–114, 2007.

[70] D. Xin, J. Han, H. Cheng, and X. Li. Answering top-k queries with multi-dimensional selections: The ranking cube approach. In *Proceedings of 2006 International Conference on Very Large Data Bases (VLDB'06)*, pages 463–475, 2006.

[71] D. Xin, J. Han, X. Li, Z. Shao, and B. W. Wah. Computing iceberg cubes by top-down and bottom-up integration: The starcubing approach. *IEEE Trans. Knowl. Data Eng.*, 19(1):111–126, 2007.

[72] D. Xin, J. Han, X. Li, and B. W. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *Proc. 2003 Int. Conf. Very Large Data Bases (VLDB'03)*, pages 476–487, Berlin, Germany, Sept. 2003.

[73] D. Xin, J. Han, Z. Shao, and H. Liu. C-cubing: Efficient computation of closed cubes by aggregation-based checking. In *Proc. 2006 Int. Conf. Data Engineering (ICDE'06)*, Atlanta, Georgia, April 2006.

[74] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *Proc. 2005 Int. Conf. Very Large Data Bases (VLDB'05)*, pages 241–252, 2005.

[75] Z. Zhang, S. won Hwang, K. C.-C. Chang, M. Wang, C. A. Lang, and Y.-C. Chang. Boolean + ranking: querying a database by k-constrained optimization. In *Proc. 2006 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'06)*, pages 359–370, 2006.

[76] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'97)*, pages 159–170, Tucson, AZ, May 1997.

[77] M. Zhu, D. Papadias, J. Zhang, and D. L. Lee. Top-k spatial joins. *IEEE Trans. Knowl. Data Eng.*, 17(4):567–579, 2005.

Vita

Dong Xin received his B. Eng degree from the Department of Computer Science and Engineering, Zhejiang University, China in 1999, and his M.S. degree in Computer Science from the same department in 2002.

His research interests include data mining, data warehousing and database systems.