SKYLINE/PREFERENCE QUERY PROCESSING

ENG PIN KWANG
(Master of Science, NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2005

ACKNOWLEDGEMENTS

The first person I would like to thank is my supervisor, Associate Professor Tan Kian Lee. I have been under his supervision and guidance as early as 1997 when I worked on my third year project. Over the years, he has taught me many things about research, especially how to craft a good research paper. I am truly grateful for his help during these years. Without his constant support and understanding, I believe I would not have reached this far today. I would also like to express my thanks to the following people: Professor Ooi Beng Chin who provides many useful suggestions and help on my Ph.D work, Dr. Chan Chee Yong whom I have many fruitful discussions for the work on evaluating skyline queries with partially-ordered domains, Dr. Barbara Catania for many invaluable suggestions for the work on pareto preference queries, Mr. Sim Hua Soon for his help with the work on numerical preference queries, Associate Professor Stan Jarzabek who has helped me a lot over the years and has inspired me to a great extent, and Dr. Anirban Mondal whose constant encouragement is a great help to me in completing my dissertation. Last of all, I would like to thank my family, especially my wife, Helen, for being understanding and patient with me throughout this period.

CONTENTS

SUMMARY

Many decision support applications are characterized by several features: (1) the query is typically based on multiple criteria; (2) there is no single optimal answer (or answer set); (3) because of (2), users are typically looking for *satisficing* answers; (4) for the same query, different users, dictated by their personal preferences, may find different answers meeting their needs. Relational database technology is ill-suited for supporting such applications because it only selects results that *exactly* match the user's criteria. Ideally, users should be able to pose *preference queries* which embed their personal preferences to the database system which then attempts to find all the *best* matches.

The need to support preference queries has recently led to the proposal of several preference frameworks for relational database systems. In this dissertation, we address performance issues associated with the implementation of features of these frameworks. Specifically, we study the evaluation of three specific types of preference queries and propose several approaches for evaluating them efficiently. All our approaches allow preference queries to be evaluated over a *large* dataset in a *limited main memory* environment. Moreover, they are *progressive* and are able to provide a *fast initial response time*.

The first type of preference queries we address is *skyline queries* which allow users to specify their preferences in terms of whether they favor low, high or different values of the attributes. We propose two online algorithms for evaluating such queries. One uses a bitmap structure while the other uses a transformation mechanism and a B^+-tree. Our performance study indicates that our second approach is superior in most cases. We also address the issue of evaluating skyline queries with partially-ordered domains. Our solution is to transform each partially-ordered attribute into a two-integer domain that allows us to exploit index-based algorithms to compute skyline queries on the transformed space. Based on this framework, we propose three novel algorithms and evaluate their performance. Our results show that our proposed techniques outperform existing approaches by a wide margin.

The second type of preference queries we address is a general form of skyline queries call *pareto queries*. Pareto queries support a wider range of base preferences and therefore allow a broader class of preferences to be specified. We propose three approaches for evaluating pareto queries. The first is a non-trivial extension of our bitmap scheme for evaluating skyline queries. The second adopts a tree structure similar to the R-tree. The third relies solely on single-dimensional indexes. The results from our performance study show that the third approach is the most attractive in terms of progressiveness and initial response time.

The third type of preference queries we address is *numerical preference queries* where preferences are specified indirectly using *scoring functions*. The scoring function is used to compute a score for each record in the database and answers are returned ordered by scores. We devise a fast partition-based query processing framework for evaluating such queries. We propose and analyze several index-based partitioning strategies. The comparative results from our performance study confirm the effectiveness of our proposed schemes.

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Motivation

1.1.1 Personalization of Database Queries

Providing personalized e-services is gradually becoming a norm in today's highly competitive environment [30]. Many companies are starting to provide personalized Business-To-Consumer (B2C) and Business-To-Business (B2B) e-services to build a closer tie with their customers and retain their loyalty. The popularity of personalized e-services can be largely attributed to the unpleasant experience encountered by users when looking for information in the World Wide Web (WWW).

Take e-procurement as an example. Electronic catalogues of large sites frequently offer millions of products for sale which necessitate the assistance of a search engine. However, many of these search engines are back-ended by relational database systems that are only capable of selecting products that *exactly* match the users' search conditions. This frequently leads to the 'no match' effect where the query comes up empty. Subsequently, users are forced to try new queries, with possibly weaker criteria such as using 'or'-conditions. However, now they get the other extreme – 'flooding' effect, where the query comes up with too many results, most of which are irrelevant to them. Such a search process is long and arduous and is a primary cause of users' frustration.

We find that most applications that exhibit such effects are typically decision support applications characterized by the following features:

1. User queries are typically based on multiple, possibly conflicting, criteria. For example, a house hunter may be interested in *cheap* houses *near* the beach. Clearly, houses near the beach are expected to be more expensive.

2. Unlike conventional applications, there may be no single *optimal* answer (or answer set). For our house hunter, it is unlikely that there exists a house that is both cheap and near to the beach. Instead, one can expect to find houses nearer to the beach to be more expensive.

3. Because of the second point, users are typically looking for *satisficing* answers that *best* match their criteria.

4. Even for the same query, different users, dictated by their personal preferences, may find different answers appealing. For example, our house hunter may be willing to pay more to be nearer to the beach. As such, it is important that all the best alternatives are presented to the users.

From the above features, we can see that relational databases are ill-suited for supporting such applications. A relational query selects only results that exactly match the user's criteria or it selects nothing. To support such applications, database systems have to be enhanced to support best match searches, personalized to individual's desires and tastes. Users should be allowed to pose personalized database queries i.e. *preference queries*, which embed their personal preferences. The database system would then attempt to find the perfect matches from the database and should there be none, all the best alternatives are automatically retrieved. Such an approach effectively combats the 'no match' and 'flooding' effects, creating an enhanced browsing experience for the users.

1.1.2 Supporting Preference Queries in Database Systems

The need to integrate preferences with database technology has not gone unnoticed by the database community. This has recently led to a widespread interest in enhancing the query capabilities of relational database systems to support preference queries. While the semantics of evaluating a standard, concrete database query is well defined i.e. extract all results that match the conditions exactly, the semantics of evaluating a preference query is still open to interpretations. However, a consensus on this issue seems to have been reached which is exemplified in several recent work such as [9], [26] and [65]. Before we formally define the problem of evaluating a preference query in a relational database system, we first define what we mean by dominance:

Definition 1.1 (Dominance). Given a set of user specified preferences P in query Q, we say a point x dominates another point y if all attribute values of x is as good as y's and at least one of them is strictly better than y's with respect to P.

Definition 1.2 (Preference Query Problem). Given a relation $R(A_1, \ldots, A_d)$ containing $|R|$ data points, a preference query Q selects a subset S of points from R that are not **dominated** by any other points in the same relation. Points in S are commonly referred to as the **maximal points** of R with respect to query Q.

The reason behind retrieving non-dominated points as results of a preference query is due to an important property of these points. Given a set of maximal points S, for any monotone scoring function $R \rightarrow \mathbb{R}$, if $p \in R$ maximizes that scoring function, then $p \in S$. In simpler terms, no matter how the user emphasizes his/her preferences, he/she can always find his/her favorites in S. In our house hunting example, no matter how our house hunter emphasizes his preferences towards price of the houses or their distances from the beach, he will always find his favorite in S. Additional, for every point $p \in S$, there exists a monotone scoring function that p maximizes. Since a scoring function represents the preferences of some user, every point in S may potentially be someone's favorite.

Intuitively, the set of maximal points, S, represents the set of satisficing answers that we describe in the previous subsection. They form the best alternatives to a user's preference query because no matter what their preferences are and how these preferences are emphasized, an appealing answer can always be found in S. Therefore, enhancing database systems with the capability to retrieve maximal points which are based on user preferences provides a means in which preference queries can be supported in these systems. How to retrieve S efficiently is the key objective of this dissertation.

The preference query problem is not something new. It is analogous to the multiobjective optimization problem well-known in operations research [87, 95]. In fact, it is really the *maximum vector problem* placed in a *database context*. The maximum vector problem is originally proposed in [72] and it involves finding all maxima of a set of points. Although the maximum vector problem has been studied earlier, solving it in the database context as the preference query problem introduces new challenges.

First, the preference queries have to be evaluated over a *large* (although finite) dataset with *limited main memory*. As discussed in [9], the algorithms for solving the maximum vector problem perform terribly in the database context because of the limited memory environment. Second, as most applications supporting preference queries are interactive in nature, it is important that first answers are returned as quickly as possible. For example, a psychology study [88] shows that users tend to accept response times of up to three seconds and will only tolerate longer runtime for more difficult tasks. Finally, if a preference query results in a large answer set, the user is unlikely to examine all the results. To conserve valuable computational resources, partial results should be returned to users initially and more results computed upon users' request. This allows users to terminate the processing prematurely as soon as they are satisfied with the partial answers, saving precious resources in computation. In other words, answers should be returned *progressively*. This is particularly important in high-traffic web sites or mobile

environments where resources are even more constrained.

The above challenges clearly signal the need for new ways of approaching the maximum vector problem in the database context. This is also the focus of this dissertation. Our goal is to devise efficient structures and algorithms that are progressive and have a fast initial response time for evaluating preference queries in relational database systems. We note that the evaluation of preference queries is potentially more expensive than hard selection relational queries because the non-monotonic nature of preferences generally leads to more complex evaluation strategies.

Before we describe our specific contributions, we shall briefly describe the type of preference queries we will be addressing in this dissertation (the respective preferences will be defined formally in the next chapter).

1.1.3 Types of Preference Queries Addressed

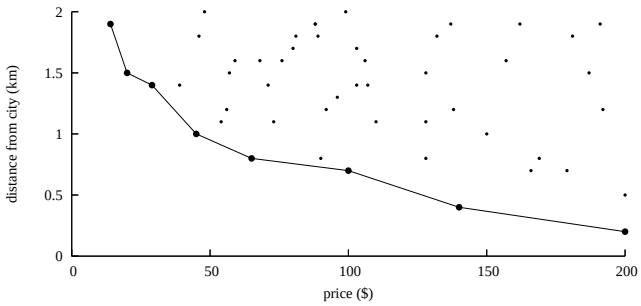
Skyline Queries

Skyline queries is first introduced in [9] and a `Skyline Of` clause is proposed as an extension to SQL. In a skyline query, users specify their preferences in terms of whether they favor low, high or different values of the attributes. All specified preferences are also regarded as equally important. The classic hotel example is shown in Figure 1.1. In this example, assume that the Hotel relation has two attributes, `Price` and `Distance`, representing the room rates of hotels and their distances to the beach respectively. Further assume that a tourist is interested in cheap hotels (preference for *low* values of `Price`) near to the beach (preference for *low* values of `Distance`).

Figure 1.1(a) shows the skyline query while Figure 1.1(b) shows the skyline of hotels. Hotels that belong to the skyline are represented by bold points that are connected in the graph. The rest of the hotels are dominated in terms of price and distance to the beach by at least one hotel that belongs to the skyline. Intuitively, those hotels that are part of the skyline are also the maximal points. As

```
SELECT *
FROM Hotels
SKYLINE OF Price MIN,
Distance MIN;
```

(a) Skyline query



(b) Skyline of hotels

Figure 1.1: Skyline example.

the example illustrates, skyline queries allow a user to specify his/her preferences directly in the query. The construct allows multiple preferences to be combined in parallel and is highly composable with relational algebra.

Pareto Queries

Although skyline queries is an important class of preference queries, they are fairly limited in expressiveness. Users are only allowed to specify whether they favor low, high or different values of the attributes. Consider our tourist example. Assume that our Hotel relation further contains two attributes, `golf_distance` and `area`, representing each hotel's distance to the nearest golf facility and the area the hotel is located respectively. Now, what if the tourist is also interested in hotels that are within 5km of a golf facility and are located in the uptown area? Such preferences cannot be expressed in a skyline query. In [65], Kießling presented a preference model for database systems and together with Köstler in [68], constructed a rich query language called Preference SQL as an extension to SQL. Kießling's preference model allows a more general form of skyline queries call *pareto queries* to be expressed. For example, our tourist can specify his new query in Preference SQL as follows:

```
SELECT     *
FROM       Hotels
PREFERRING LOWEST(price) AND LOWEST(distance)
           AND golf_dist BETWEEN [0,5] AND
           area IN ('uptown');
```

The pareto query allows a set of *base preferences* such as LOWEST, BETWEEN and IN to be directly specified on various attributes and it combines them through an AND operator to signify that all the constituent preferences are equally important. As our example illustrates, pareto queries can potentially cover a broader class of preference queries.

Numerical Preference Queries

In numerical preference queries, preferences are specified indirectly using *scoring functions*. A user expresses his/her preferences by assigning weights to various attributes of interest. The system then computes a score for each data point according to some specific function of the given weights. A data point x is more preferred than another point y if x has a higher score than y. For example, our tourist might view the price of hotels and their distances to the beach as more important compared to the rest of the attributes. Hence, he assigns a weight of 0.4 each to both attributes while setting equal and low weights to the rest. The system then computes the score for each hotel and output the top scoring ones.

An analogous approach has also been applied for multiobjective optimization problems in operations research where a multiobjective problem is first transformed into a mono-objective one before solving. In the context of our preference query problem, one can imagine that all attributes of interest are "compressed" into a single representative attribute, the score. Therefore, there is only one attribute for comparison and a point x dominates another point y if x has a higher score than y. The maximal points are, thus, the set of points with the highest scores.

In practice, this set of maximal points are generally too small a set to choose from. This lead to the proposal of the *top-k* query model where the best k answers are returned. Since k is user-definable, this might result in some non-maximal points being returned. Nonetheless, numerical preference queries are popular in several database and information retrieval applications, especially those that require multi-feature or full-text searches.

1.2 Contributions

In this dissertation, we address the issue of computational efficiency of the preference query problem. We adopt the preference framework presented in [65] and propose several algorithms and data structures that are suitable for answering preference queries efficiently in relational database systems. All our approaches are completely progressive and provide a fast initial response time by returning answers as soon as they become available. The specific contributions are as follows:

1. We propose two techniques for solving skyline queries. The first technique, called Bitmap, is completely non-blocking and exploits a bitmap structure to quickly identify whether a point belongs to the skyline or not. The second technique, called Index, exploits a transformation mechanism and a B^+-tree index to return skyline points in batches. All the techniques are implemented and an extensive performance study is conducted to compare their performance against three existing algorithms. Our experimental studies show that Index is superior in most cases while Bitmap performs well for small number of distinct values per attribute as well as for large number of skyline points.

2. We study the evaluation of skyline queries with partially-ordered attributes. Because such attributes lack a total ordering, traditional index-based evaluation algorithms that are designed for totally-ordered attributes can no longer prune the space as effectively. Our solution is to transform each partially-ordered attribute into a two-integer domain that allows us to exploit index-based algorithms to compute skyline queries on the transformed space. Based on this framework, we propose three novel algorithms. We implemented the proposed schemes and evaluated their performance. Our results show that our proposed techniques outperform existing approaches by a wide margin (between a factor of 2 and 16). To the best of our knowledge, this is the first work that examines the problem of evaluating skyline queries with partially-ordered domains.

3. We propose three approaches for evaluating pareto queries. The first approach is a non-trivial extension of our Bitmap technique for solving skyline queries. Taking advantage of the fact that bitwise operation is fast, this approach can achieve a high level of efficiency. However, because bitmap indexes incur a relatively high storage and maintenance cost in practice, the application of this approach is limited to static databases such as data warehouses where updates are rare and queries frequent. To deal with dynamic databases, we propose the second approach which is a tree structure similar to the R-tree. By sacrificing some efficiency, it provides a space efficient solution with a lower maintenance cost compared to the first approach. While these two approaches are essentially multi-dimensional indexes, our third approach is based only on **single-dimension indexes** such as the B^+-tree. While it requires a single dimensional index to be built for each attribute, these indexes are still comparably cheaper to maintain than a single multi-dimensional index. Moreover, since most commercial DBMS support at least one type of single-dimensional index, this approach can be easily integrated into existing database systems. We also conducted an extensive experimental study on the effectiveness of our three approaches against existing techniques. Our results indicate that the last approach is the most attractive in terms of progressiveness and initial response time.

4. We devise a fast partition-based query processing framework for evaluating numerical preference queries. We propose and analyze several index-based partitioning strategies. Index-based approaches are attractive because most indexes inherently partition the databases and hence save the cost to partition the database at runtime. We also performed an extensive experimental study to compare the performance of our algorithm against existing approaches. The comparative results confirm the effectiveness of our proposed schemes.

1.3 Thesis Outline

The thesis is organized as follows:

- Chapter 2 describes the preference framework that we adopt in this dissertation. The framework is proposed by Kießling in [65]. It is a semantically rich preference model for database systems, based on preferences as strict partial orders. Related work are also reviewed. These related work can generally be divided into two groups with one group adopting a *qualitative* approach while the other adopting a *quantitative* approach to supporting preference queries. Other related approaches to supporting user preferences are also described.

- Chapter 3 presents our two approaches to evaluating skyline queries. We describe how the skyline is computed progressively using our data structures and algorithms and address several important issues pertaining to our approaches. Results from the performance study conducted to evaluate the effectiveness of our algorithms are also presented.

- Chapter 4 presents our framework for evaluating skyline queries with partially-ordered domains. Based on the framework, we describe three novel algorithms for evaluating skyline queries involving partially-ordered attribute domains. Results from the performance study conducted to evaluate the effectiveness of the algorithms are also presented.

- Chapter 5 presents our three approaches to evaluating pareto queries. We describe the data structures and algorithms used in the various approaches. We prove the correctness of our algorithms and analyze their performance through an extensive experimental study. Findings based on the study are also presented.

- Chapter 6 presents our partition based framework and algorithm for evaluating numerical preference queries. We describe salient features of our framework that enable such queries to be processed efficiently. We also analyze

several index-based partitioning strategies. Results from our experimental studies showing the effectiveness of our framework are also presented.

- We conclude our work in Chapter 7 with a summary of our contributions. We also provide directions for future work.

We note that a preliminary version of Chapter 3 was published in the Proceedings of the 27th International Conference on Very Large Data Bases, Roma, Italy, in September, 2001 [97]. Its expanded version was published in Data & Knowledge Engineering, Volume 46, Number 2, in 2003 [34]. A preliminary version of Chapter 4 was accepted for publication in the Proceedings of the 21st International Conference on Data Engineering, Tokyo, Japan, in April, 2005 [17]. Its expanded version was accepted for publication in the Proceedings of the 24th ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, in June, 2005 [18]. A preliminary version of Chapter 6 was published in the Proceedings of the 19th International Conference on Data Engineering, Bangalore, India, in March, 2003 [32]. Expanded versions of Chapters 5 and 6 are currently under review [33, 35].

CHAPTER 2

Preliminaries

In this chapter, we first describe the preference framework that we have adopted for our work. The framework is proposed in [65] for database systems. It is a powerful framework based on modeling preferences as strict partial orders. We give an overview of the framework, focusing only on those aspects of the framework that are relevant for the scope of our work. Full details can be found in [65]. The rest of the chapter will then review related work which are grouped according to the approach they use to support preference queries.

2.1 A Preference Framework for Relational Database Systems

2.1.1 Preferences

Consider a dataset D having a set of attributes $A = \{A_1, A_2, \ldots, A_k\}$. Let the domain of A_j be $dom(A_j)$ for $1 \leq j \leq k$ and $dom(A) = \times_{A_j \in A} dom(A_j)$.

Definition 2.1 (Preference). A preference on the set of attributes A is defined as $\mathrm{P} = (A, <_P)$ where $<_P \subseteq dom(A) \times dom(A)$ is a strict partial order.

Hence, $<_P$ is irreflexive, anti-symmetric and transitive. Following the notation in [65], we write "$x <_P y$" to mean y is *more preferred* than x for $x, y \in \mathrm{dom}(A)$ although we occasionally use the term "*y dominates x*" for the same meaning. Also, if x and y are *unranked*, we mean $\neg(x <_P y) \wedge \neg(y <_P x)$.

The preference P can also be represented as a directed acyclic graph G called **better-than graph**. Several quality notions can be defined between two values x and y in G: we say y is a predecessor of x if $x <_P y$. If y has no predecessor, it is *maximal* and defined to be at *level 1*. Otherwise, y is at level j if the longest path between y and a maximal value has $j - 1$ edges. Finally, we say x and y are unranked if there is no directed path between them. These quality notions are used for unordered domains. For ordered domains, a continuous distance function is used instead of the discrete level function to distinguish the levels.

The maximal values of a preference P forms the results of a preference query and is defined formally as follows:

Definition 2.2 (Maximal values). The maximal values of P = $(A, <_P)$ is given by the set $\{v \in dom(A) \mid \neg\exists w \in dom(A) : v <_P w\}$.

For specifying preferences, [65] promotes a *constructor-based* approach where constructors, acting as *preference templates*, are used to instantiate the various types of preferences. These constructors are further divided into **base** and **complex** constructors. A base constructor instantiates a base preference and is classified as *non-numerical* or *numerical*, depending on whether the domain of the attributes under consideration is unordered or ordered respectively. On the other hand, a complex constructor inductively creates complex preferences by combining the base preferences. In [65], a pre-defined set of constructors are proposed. They are considered valuable for personalized searches based on practical experiences gained from e-shopping applications [68]. New constructors can be added as required by the application domain. We describe these constructors next.

2.1.2 Base Preference Constructors

The base constructors are divided into two sets, one for each type of domain. We shall repeat the formal definitions from [65] here and illustrate them with examples. For ease of illustration, we shall use the sample hotel relation shown in Table 2.1.

Id	Rates	Area	Stars
1	280	midtown	2
2	190	uptown	3
3	308	midtown	3
4	314	midtown	4
5	257	uptown	2

Table 2.1: Hotels relation.

The attributes of the relation consist of the hotel's `id`, the starting `rates` of a room, the `area` the hotel is in and the ratings of the hotel represented by the number of `stars`. The domain of the attribute `area` is {uptown, midtown, downtown} while the domain of other attributes are integers.

Non-numerical Base Preference Constructors

POS preference: P := POS(A, POS-set = {$v_1, \ldots, v_m$})

FORMAL: *Let POS-set $\subseteq$ dom(A) be finite. P is a POS preference if $x <_P y$ iff $x \notin$ POS-set $\wedge$ $y \in$ POS-set.*

INTUITION: A POS preference specifies that some desired values *should* be from a set of favorites $v_1, \ldots, v_m \in dom(A)$, called *positive* values. If this is not possible, instead of returning nothing, any *other value* from $dom(A)$ is returned.

EXAMPLE: POS(`area`, {downtown, midtown}). Hotels 1, 3 and 4 are returned as the answers since they are located in midtown (which is in the POS-set).

NEG preference: P := NEG(A, NEG-set = {$v_1, \ldots, v_m$})

FORMAL: *Let NEG-set $\subseteq$ dom(A) be finite. P is a NEG preference if $x <_P y$ iff $y \notin$ NEG-set $\wedge$ $x \in$ NEG-set.*

INTUITION: A NEG preference specifies that the desired values *should not* be from a set of dislikes $v_1, \ldots, v_m \in dom(A)$, called *negative* values. If this is not possible, instead of returning nothing, any disliked value is returned.

EXAMPLE: NEG(`area`, {uptown}). For this NEG preference, uptown is the disliked value. Hence, hotels 1, 3 and 4 are returned as the answers since the area they are located in are not in the NEG-set.

POS/NEG preference: P := POS/NEG(A, POS-set = {$v_1, \ldots, v_m$}; NEG-set = {$v_{m+1}, \ldots, v_{m+n}$})

FORMAL: *Let POS-set, NEG-set ⊆ dom(A) be finite and disjoint. P is called a POS/NEG preference if $x <_P y$ iff ($x \in$ NEG-set $\wedge$ $y \notin$ NEG-set) $\vee$ ($x \notin$ NEG-set $\wedge$ $x \notin$ POS-set $\wedge$ $y \in$ POS-set).*

EXAMPLE: POS/NEG(`area`, {downtown}; {uptown}). POS/NEG preference is a combination of the previous preferences. Thus, hotels 1, 3 and 4 are the answers as the area they are located in do not belong to the NEG-set.

POS/POS preference: P := POS/POS(A, POS1-set = {$v_1, \ldots, v_m$}; POS2-set = {$v_{m+1}, \ldots, v_{m+n}$})

FORMAL: *Let POS1-set, POS2-set ⊆ dom(A) be finite and disjoint. POS-1 set are the favorite values, POS2-set are the second-best alternatives. P is called POS/POS preference, if $x <_P y$ iff ($x \in$ POS2-set $\wedge$ $y \in$ POS1-set) $\vee$ ($x \notin$ POS1-set $\wedge$ $x \notin$ POS2-set $\wedge$ $y \in$ POS2-set) $\vee$ ($x \notin$ POS1-set $\wedge$ $x \notin$ POS2-set $\wedge$ $y \in$ POS1-set).*

EXAMPLE: POS/POS(`area`, {downtown}; {uptown}). A POS/POS preference allows the specification of optimal values (POS1-set) and alternative values (POS2-set). Since there are no hotels in downtown, hotels 2 and 5 which are the second-best alternatives are returned as the answers.

EXPLICIT preference: P := EXPLICIT(A, explicit-graph{$(v_1, v_2), \ldots$})

FORMAL: *Let explicit-graph = (v_1, v_2), ... represents a finite acyclic better-than graph, where $v_i \in$ dom(A). Let V be the set of all v_i occurring in the explicit-graph. We induce a strict partial order S = (V, $<_S$) on V as follows: each pair (v_i, v_j) $\in$ the explicit-graph means that $v_i <_S v_j$, and $v_i <_S v_j \wedge v_j <_S v_k$ implies $v_i <_S v_k$. P is an EXPLICIT preference if for $x, y \in$ dom(A), $x <_P y$ iff $x <_S y \vee$ ($x \notin$ range($<_S$) $\wedge$ $y \in$ range($<_S$)).*

EXAMPLE: EXPLICIT(`area`, {(uptown, downtown), (midtown, uptown)}. An EXPLICIT preference allows the explicit specification of any better-than relationships. In this query, the user prefers downtown to uptown and uptown to midtown. The EXPLICIT-graph is: *downtown* (*level* 1) $\rightarrow$ *uptown* $\rightarrow$ *midtown* (*level* 3).

Thus, hotels 2 and 5 are returned as the answers since they are in uptown (as compared to the rest of the hotels which are in the midtown area).

Numerical Base Preference Constructors

AROUND preference: P := AROUND(A, z)

FORMAL: *Given a value $z \in dom(A)$, $\forall v \in dom(A)$ we define distance(v, z) = abs$(v-z)$. P is an AROUND preference if $x <_P y$ iff distance$(x, z) >$ distance(y, z). If distance(x, z) = distance(y, z) and $x \neq y$, then x and y are unranked.*

EXAMPLE: AROUND(`rates`, 250). AROUND preference favors values that are close to a target value. Here, hotel 5 is the answer as its rate is nearest to 250 in terms of the distance function.

BETWEEN preference: P := BETWEEN(A, [low,up])

FORMAL: *Given an interval [low, up] $\in dom(A) \times dom(A)$, low $\leq$ up, $\forall v \in dom(A)$, we define distance(v, [low, up]) = if $v \in$ [low, up] then 0 else if $v <$ low then low - v else v - up. P is a BETWEEN preference if $x <_P y$ iff distance(x, [low, up]) > distance(y, [low, up]). If distance(x, [low, up]) = distance(y, [low, up]) and $x \neq y$, then x and y are unranked.*

EXAMPLE: BETWEEN(`rates`, [200, 220]). A BETWEEN preference prefers values close to some specified interval. Thus, hotel 2 is returned as the answer as it is nearest to the minimum preferred rate of 200 in terms of the distance function.

LOWEST, HIGHEST preference: P := LOWEST(A), HIGHEST(A)

FORMAL: *P is called LOWEST preference if $x <_P y$ iff $x > y$ while P is called HIGHEST preference if $x <_P y$ iff $x < y$, for $x, y \in dom(A)$.*

EXAMPLE: LOWEST(`stars`) and HIGHEST(`stars`) return hotels $\{1, 5\}$ and 4 as answers because these are hotels with the lowest and highest ratings respectively.

SCORE preference: P := SCORE(A, f)

FORMAL: *Given a scoring function f: $dom(A) \rightarrow \mathbb{R}$. If '<' is the familiar 'less-than' order on $\mathbb{R}$, then P is a SCORE preference if for x, $y \in dom(A)$: $x <_P y$ iff $f(x) < f(y)$.*

EXAMPLE: Assume a scoring function $f = 0.1 \times$ `rates`, then SCORE(`rates`, f) will return hotel 4 as the answer because it gives the highest score.

2.1.3 Complex Preference Constructors

A complex preference is inductively combined from the base preferences using complex preference constructors. In [65], several complex preference constructors are proposed. However, for the scope of this dissertation, we will only consider a subset of these constructors. For ease of presentation, the definitions are based on two preferences (generalization to more preferences is straightforward). Moreover, we shall assume that all the constituent preferences in a complex preference are declared on disjoint sets of attributes.

Pareto preference: P := P1 $\otimes$ P2

FORMAL: *Assume two preferences P1 = (A1, $<_{P1}$) and P2 = (A2, $<_{P2}$). For $x = (x_1, x_2)$ and $y = (y_1, y_2) \in dom(A1) \times dom(A2)$, we define $x <_{P1 \otimes P2} y$ iff $(x_1 <_{P1} y_1 \wedge (x_2 <_{P2} y_2 \vee x_2 = y_2)) \vee (x_2 <_{P2} y_2 \wedge (x_1 <_{P1} y_1 \vee x_1 = y_1))$.*

INTUITION: A pareto preference considers all its constituent preferences as **equally important** preferences. Hence, for a data tuple t_1 to *dominate* another tuple t_2, it cannot be worse than t_2 for any attributes.

EXAMPLE: HIGHEST(`stars`) $\otimes$ AROUND(`rate`, 200) $\otimes$ POS(`area`, {uptown}). Hotels 2 and 4 are returned as the answers. Hotel 1 is dominated by hotels 2 and 5 because all its attribute values are either equal or worse than hotels 2 and 5 with respect to the pareto preference. Hotels 3 and 5 are dominated by hotel 2. Although hotel 3 has the same ratings as hotel 2, its rates are much higher than the ideal 200 than hotel 2 and its location is not in one of the favored areas. For hotel 5, although it is located in the uptown area, it ratings and rates are both worse than hotel 2 with respect to the HIGHEST and AROUND preferences respectively.

Grouped preference: P groupby A

FORMAL: *Let preference P = (A1, $<_P$). For $x = (x_1, x_2)$ and $y = (y_1, y_2) \in dom(A1) \times dom(A2)$, we define $x <_{P\ groupby\ A2} y$ iff $x_1 <_P y_1 \wedge x_2 = y_2$.*

A grouped preference P is evaluated in *grouped mode* where grouping is done with respect to equal A2-values. One common use of grouped preferences is to specify skyline queries [9] which we will describe next.

Skyline preference: P := (P1 $\otimes$ P2) groupby DIFF

FORMAL: *Follows from the definitions of pareto and grouped preferences where P1 and P2 is each restricted to either a LOWEST or a HIGHEST base preference and DIFF represents the set of attributes (may be $\emptyset$) for which grouping is to be done.*

INTUITION: Without the DIFF attributes, the skyline preference is simply a pareto preference consisting of only HIGHEST and LOWEST base preferences. The inclusion of DIFF attributes means that we want to retrieve the best values (based on the HIGHEST and LOWEST base preferences) with respect to each distinct value of the DIFF attributes.

EXAMPLE: LOWEST(`rates`) groupby `stars`. There are three groups: two 2-stars hotels, two 3-stars hotels and one 4-stars hotel. Among the 2-stars hotels, hotel 5 dominates hotel 1 because it has lower rates. Among the 3-stars hotels, hotel 2 dominates hotel 3 because it has lower rates and there is no comparison for 4-stars hotels. Hence, hotels 2, 4 and 5 are returned as the answers.

Numerical preference: P := rank(F)(P1, P2)

FORMAL: *Assume that P1 := SCORE(A1, f1), P2 := SCORE(A2, f2) and F is a* ***combining function****: $\mathbb{R}\times\mathbb{R} \rightarrow \mathbb{R}$. If '$<$' denotes the familiar 'less-than' order on $\mathbb{R}$, then for $x = (x_1, x_2)$ and $y = (y_1, y_2) \in dom(A1) \times dom(A2)$, $x <_{rank(F)(P1,P2)} y$ iff $F(f1(x_1), f2(x_2)) < F(f1(y_1), f2(y_2))$.*

EXAMPLE: Let us assume that the `rates` and `stars` attribute values in Table 2.1 are range normalized as follows. Let min_A and max_A be the minimum and maximum values of some attribute A respectively. We assume that min_A and max_A are captured in the DBMS's catalog. Then, we can range normalize a value v of A by $\frac{max_A - v}{max_A - min_A}$. In this example, we shall assume that the maximum and minimum values of attributes `rates` and `stars` are the maximum and minimum values of the respective attributes in Table 2.1. Normalized values of attributes `rates` and `stars`

are shown in Table 2.2. Let P1 := SCORE(`rates`, f_1) where $f_1 = 0.2 \times v_1, v_1 \in$ dom(`rates`) and P2 := SCORE(`stars`, f_2) where $f_2 = 0.8 \times v_2, v_2 \in$ dom(`stars`). Let F = SCORE(`rates`, f_1) + SCORE(`stars`, f_2). The scores computed using F for the sample relation is also shown in Table 2.2. From the table, we can see that hotel 4 has the highest score and is returned as the answer.

Id	Rates	Area	Stars	Score
1	0.27	midtown	0.0	0.05
2	1.00	uptown	0.5	0.60
3	0.05	midtown	0.5	0.41
4	0.00	midtown	1.0	0.80
5	0.46	uptown	0.0	0.09

Table 2.2: Hotels relation with normalized values.

2.1.4 The Best-Matches-Only (BMO) Model

In [65], the evaluation of preference queries is based on a Best-Matches-Only (BMO) query model and works conceptually as follows [51]:

- Given a preference query Q, find all *perfect matches* from the relation with respect to the preferences specified in Q.
- If no perfect matches exist for Q, retrieve all *best matching alternatives*, but *nothing worse*.

In the BMO model, maximal values are retrieved from the relation. Since these values may not be perfect matches, *query relaxation* is implicit in the evaluation. Moreover, all non-maximal values are also eliminated during the evaluation. Hence, only the best matching answers are retrieved. It is not difficult to see that the BMO query model is an excellent candidate for solving the preference query problem and we adopt its operational semantics throughout this dissertation.

2.2 Related Work

There are two main approaches to handling preferences in the context of database queries. In the *qualitative* approach, preferences between tuples are typically expressed directly using binary preference relations. These approaches provide ways of composing preferences with other query constructs and extend the query semantics to handle preferences. In the *quantitative* approach, preferences are reflected indirectly using scoring functions. A scoring function associates a numeric score to each tuple where a high score would mean that the tuple is more preferred and best matches the user's preferences. However, we note that scoring functions are generally less expressive and thus, this approach is limited in its applicability. We shall now present related work pertaining to both approaches as well as work that are generally related to supporting user's preferences but which do not fall in any of the two approaches.

2.2.1 Qualitative Approach

Frameworks and Query Languages

One of the earliest work on qualitative preference queries is [73]. The authors propose an extension to the domain relational calculus [74] to express preferences in queries. A `prefer` clause is introduced to qualify a standard database query. For example, a user can specify: *Given a set of tuples that satisfy condition C, prefer those satisfying condition P.* Clearly, C represents the hard constraints while P is the user's preferences. Its operational semantics is to first evaluate the query based just on condition C. Then, the prefer clause i.e. P is applied to the answers. If application of P results in no answers, the prefer clause is ignored and all answers prior to the application of P are returned. Otherwise, results satisfying both C and P are returned.

[73] also addresses the issue of composability by illustrating how preferences can be combined and prioritized The authors also discuss how to handle maxi-

mum/minimum value preferences through aggregate functions. However, as each preference condition is evaluated in a boolean manner, this approach is limited in expressiveness. Moreover, efficiency issues for the evaluation and optimization of preference queries are also not addressed.

A recent preference framework similar to Kießling's framework [65] is independently proposed by Chomicki [25, 26]. Chomicki introduces a logical framework having the same view of preferences as strict partial order, but expressing preferences more generally as logical formulas. He studies various classes of these formulas and proposes a relational operator called *winnow* for retrieving the preferred tuples. Winnow offers a declarative semantics and is used for composing preference relation i.e. the order the preference induces over the potential answers, with relational algebra. Although sharing the view that preferences should be strict partial order, Chomicki has also found useful preferences that violate the order and has proposed ways of accommodating them.

Although winnow is a rich model, it can be complex to understand how to express preferences and compose them in this model. Moreover, it is not clear how an efficient implementation of winnow can be realized in a relational system as no practical implementation like Preference SQL [68] is given. Only the scope of various potential algorithms suitable for implementing winnow is examined in [26].

Besides Chomicki's framework, several work [44, 45, 67, 70] also adopt a logical approach to preferences. However, they are studied in the context of deductive databases. These related work all propose extending Datalog with clausally-defined preference relations and describe declarative and operation semantics for these extensions. Although techniques that are proposed from these work can potentially be applied to the relational data model, the adaption is challenging as most of them either adopt special evaluation mechanisms or the preferences addressed are limited only to rule priorities.

Evaluating Skyline Queries

The skyline operator is first proposed in [9] for evaluating skyline queries in relational database systems. The operator has a clear partial-order semantics and is composable with relational algebra. Many recent work has proposed efficient, secondary-memory and relationally well-behaved algorithms for evaluating skyline queries. We review them in this subsection. Throughout our discussion of the algorithms, we shall consider our data model as consisting of a set of data tuples. However, we will occasionally consider this set of data tuples as a set of multi-dimensional points whenever it makes the discussion clearer. Furthermore, we shall assume that the skyline queries consist of only LOWEST preferences i.e., the users favor low values for each attribute or dimension of the dataset.

Block Nested Loop (BNL) Algorithm [9]

The block nested loop algorithm is an iterative algorithm that repeatedly scans a set of tuples. In each iteration, a *window* of incomparable tuples are kept in the main memory. When a tuple p is read from the input relation, p is compared with the tuples in the window. There are three possible outcomes:

1. If p is dominated by a tuple in the window, it means that p cannot be in the skyline. Thus, p is discarded.

2. If p dominates one or more tuples in the window, these tuples are eliminated (since they cannot be in the skyline), and p is inserted into the window.

3. If p is incomparable with all tuples in the window (i.e., it neither dominates nor being dominated), it is either inserted into the window if there is sufficient room in the window, or written to a temporary file on disk.

At the end of each iteration, those tuples in the window that have been compared against all tuples that have been written out to the temporary file are certain to be in the skyline, and hence can be returned to the user (and removed from the

window). In the next iteration, the algorithm will proceed in the same manner with the remaining tuples in the window and the temporary file as the input relation.

One of the most expensive process in the algorithm is the time spent to compare a tuple with the tuples in the window. To speed up these comparisons, the tuples in the window are organized as a *self-organizing list*. When a tuple q of the window is found to dominate another tuple, then q is moved to the *beginning* of the window. In this way, tuples that are highly dominant will float to the top of the window, and subsequent input tuples will be compared against them first.

The key strength of the algorithm is its wide applicability as it does not require the data to be indexed or sorted. Its key weakness is its lack of progressiveness. It is also not attractive in terms of producing fast initial response time as it requires at least one pass over the input relation before a set of results can be identified.

Divide and Conquer (DC) Algorithm **[9]**

The DC algorithm is an extension of the basic two-way divide and conquer algorithm for computing maximal vectors [72, 89]. The algorithm employs a m-way partitioning strategy for computing skylines and works as follows.

- Compute the α-quantiles of a set of input tuples along a certain attribute. Split the tuples into m partitions such that each partition fits in the memory. Let these partitions be $P_1, \ldots, P_m$.

- Compute the skyline S_i of partition P_i, $1 \leq i \leq m$, using any known main memory algorithm.

- Compute the overall skyline as the result of merging S_i pairwise. Within the merge operation, m-way partitioning is applied so that all sub-partitions can be merged in main memory. We note that the recursive applications of the m-way partitioning pick different attributes for splitting. It is also interesting to note that some pairs of merging can be skipped as tuples within these pairs are already incomparable.

An example of the merge step using 2-way partitioning is illustrated in Figure 2.1 using the skyline example from Figure 1.1. The dataset has been partitioned into four sub-partitions using the median of the price and distance attributes. The local skyline of each sub-partition are represented by connected bold points while the final skyline is represented by those bold points with enclosing square boxes in the graph. Notice that we do not need to merge sub-partitions $P_{1,2}$ and $P_{2,1}$

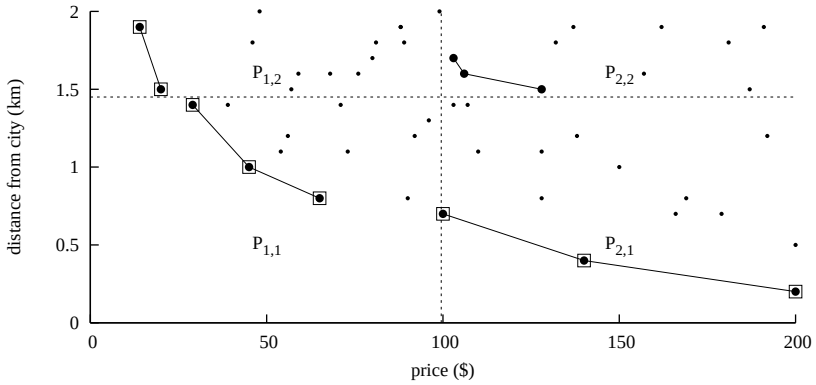


Figure 2.1: Merge step of the divide and conquer algorithm.

because tuples of these two sub-partitions are incomparable. Sub-partitions $P_{1,1}$ is first merged with $P_{2,1}$ to give R_1. Then, $P_{1,2}$ is merged with $P_{2,2}$, resulting in R_2. Finally, $P_{1,1}$ is merged with R_2 resulting in R_3. The resulting set of tuples is given by $R_1 \cup R_3$. Note that the merging is *recursive* i.e. merging two sub-partitions will require both partitions to be further partitioned and the merging terminates when all relevant attributes have been considered or one of the partitions is empty or contains only one tuple.

Like the block nested loop scheme, the divide and conquer technique cannot produce the skyline progressively. Moreover, it is not expected to perform well for small memory systems as it requires the partitions to be in-memory.

The Sort-Filter-Skyline (SFS) Algorithm [27]

In [27], Chomicki et. al. propose a variation of the block nested loop algorithm which works as follows. First, the dataset is sorted using a monotone preference function over the skyline attributes. Subsequent processing steps is similar to the

block nested loop algorithm except for three key differences: 1) tuples are inserted into the window in ascending order of their scores, 2) instead of checking whether a candidate tuple dominates a tuple in the window and vice versa, it is only necessary to check the latter case, and 3) if a candidate tuple is not dominated by any tuples in the window and no tuples has been written to the temporary file yet, then the candidate tuple is guaranteed to be in the skyline and can be output immediately.

The last difference allows the sort-filter-skyline algorithm to exhibit some form of progressiveness (the progressiveness is paused once the window is full). Moreover, for a complete skyline, at least one scan of the dataset is required as it is possible that the last tuple (the one with the highest score) is in the skyline.

***Using B-trees* [9]**

The computation of the skyline can also be facilitated by index structures. In [9], a method based on B-tree is described. Assuming that each tuple has j attributes and there is an index for every attribute, the skyline can be computed as follows.

- Scan all the indexes *simultaneously* i.e. the first entry from each index is retrieved first followed by the second entry from each index and so on. This continues until a *match* occurs. The match represents the first tuple whose id is seen by all the indexes during the scan.

- Let $a_1, a_2, \ldots, a_j$ be the attribute values of the first match. For each index i, it is scanned further to retrieve additional entries that have the same value as a_i, for $1 \leq i \leq j$. This may result in further matches. Next, compare the matches pairwise and remove those that are dominated. The remaining matches are definitely part of the skyline and can be returned immediately, providing a fast initial response time.

- Scan the index entries of the first attribute's index. If a tuple has not been seen before (i.e., the index entries of this tuple have not been examined prior to the first match), it is definitely not in the skyline and can be eliminated. If

any of the indexes contain an index entry to this tuple prior to the first match, then the tuple may or may not be in the skyline. To determine whether it is in the skyline, an existing skyline algorithm can be applied.

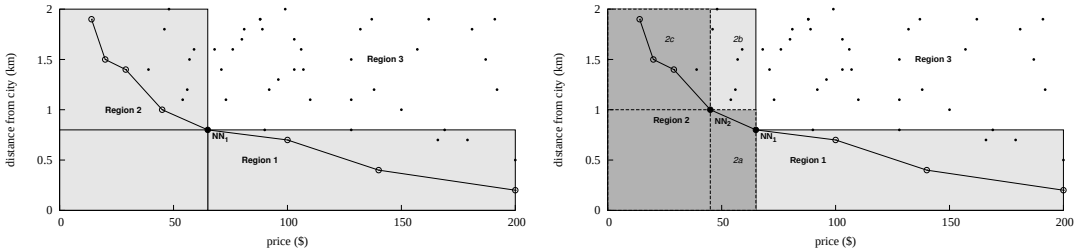
A critical factor that will affect the performance of this algorithm is how fast the first match can be found. If a match is found late (which is likely to be the case for large number of attributes), it will result in a high initial response time. Furthermore, since a large number of index entries would have already been examined, a larger number of tuples will have to be retrieved and processed using the alternative algorithm, reducing its overall efficiency. Nevertheless, we can expect this algorithm to perform well in general, when the skyline is small and the first match can be found quickly.

The Nearest Neighbor (NN) Algorithm **[69]**

NN is a divide and conquer algorithm that uses nearest neighbor searches for computing the skyline. The algorithm starts by computing the nearest neighbor of an ideal point e.g. the origin of the data universe. The authors propose using a multi-dimensional index such as the R-tree [50]. As proven in [69], this nearest neighbor is guaranteed to be in the skyline. It is then used to partition the search space such that processing is required only on j regions where j is the number of dimensions. Subsequent processing is carried in a similar way on each region i.e. find the nearest neighbor of that region and then use it for partitioning the region into sub-regions. This continues until all regions have been investigated.

As an example, consider the skyline query in Figure 1.1. The algorithm starts by searching for the nearest neighbor of the origin using some monotonic distance function. This is illustrated in Figure 2.2(a). The nearest neighbor is NN_1. Let (n_x, n_y) denotes the coordinates of NN_1. NN_1 splits the data space into three regions. Let a region be denoted by $[x_1, x_2)$ $[y_1, y_2)$ where (x_1, y_1) and (x_2, y_2) denote the coordinates of the lower left and upper right corners of the region respectively. Since region 3, $[n_x, \infty)$ $[n_y, \infty)$, contains points having greater or

equal x and y values than NN_1, they are dominated by NN_1 and hence need not be considered further. Only region 1 = $[0, \infty)$ $[0, n_y)$ and region 2 = $[0, n_x)$ $[0, \infty)$ can possibly contain skyline points and hence they will be processed next. Assume



(a) First step of algorithm (b) Second step of algorithm

Figure 2.2: The NN algorithm.

that region 2 is processed next. Again, we look for the nearest neighbor of this region to the origin. In Figure 2.2(b), it is the point NN_2. Following the same strategy as the previous step, NN_2 is used to partition region 2 into sub-regions and further processing carried on sub-regions 2a and 2c. A region is not processed further if it is empty. This continues until all skyline points are found.

As noted in [69], NN partitioning for more than 2 dimensions may result in duplicate skyline points to be found. The authors propose several ways to eliminate these duplicates. Nonetheless, the presence of duplicates means that certain access paths are traversed multiple times, resulting in redundant work. Moreover, as shown in [85, 86], the NN algorithm also has a large space overhead.

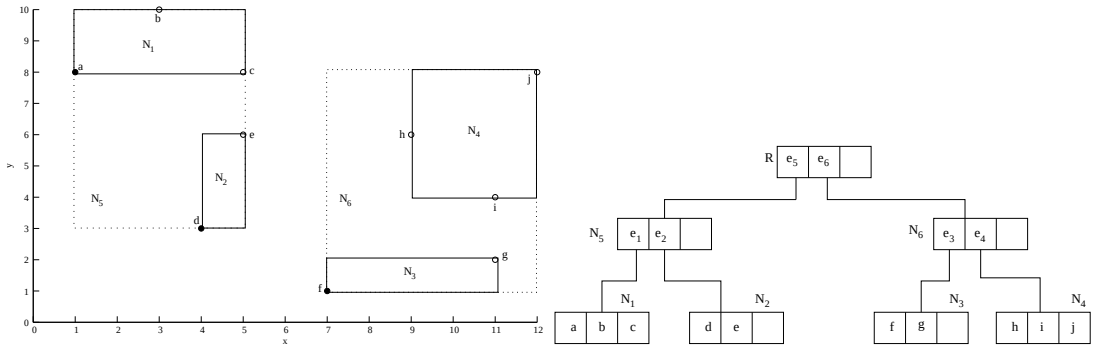
The Branch and Bound Algorithm (BBS) [85, 86]

In [85], the authors adopt the idea of using nearest neighbors for skyline computation and propose a branch and bound algorithm (BBS) that is I/O optimal (hence, avoids the need to handle duplicates) and has a smaller space overhead than NN. Similar to NN, the data is assumed to be indexed by R-trees (although the algorithm is applicable to any multi-dimensional index).

The algorithm starts by inserting all entries of the root node of the R-tree into a heap H. Entries in the heap are sorted in ascending order of their distances

to the origin. For an intermediate entry, coordinates of its lower left corner are used for computing the distance while for a data point, its actual dimensional values are used. The top entry of H is then removed for further investigation. If it is dominated by any skyline points, it would be discarded. Otherwise, for an intermediate entry, it would be expanded and its child entries are added to the heap if they are also not dominated by any existing skyline points. For a data entry, it is guaranteed to be a skyline point and can be output immediately. All found skyline points are kept in a list S for subsequent checking. The next entry from the heap is then picked and similarly processed. This continues until the heap contains no entries, which terminates the algorithm.

As an example, consider Figure 2.3. Figure 2.3(a) shows the dataset we used for this example (ten points, labelled a to j) and Figure 2.3(b) shows the corresponding R-tree with node capacity 3. An intermediate entry e_i represents the minimum bounding rectangle (MBR) of a lower level node and the leaf entries store the data points. Assume that the distances are computed using the L_1 norm i.e. the distance



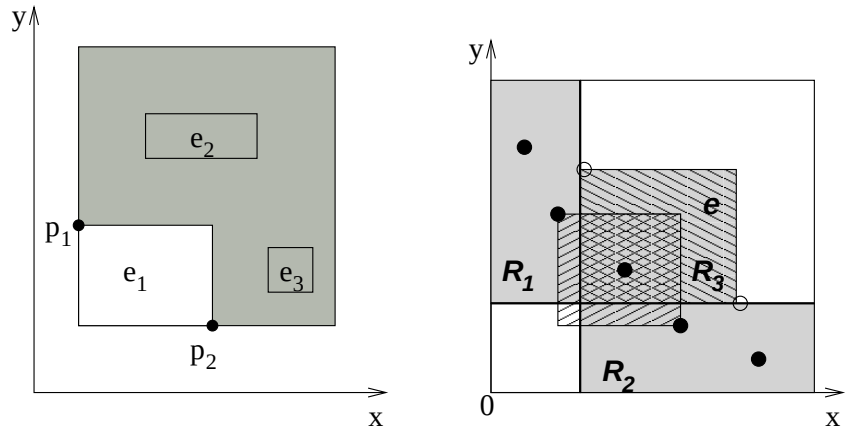
(a) Sample dataset. (b) R-tree for dataset.

Figure 2.3: The BBS algorithm.

of a data point is the sum of its coordinates. For an intermediate entry, the distance is the sum of its lower-left corner point's coordinates. The algorithm starts from the root node of the R-tree and inserts entries (e_5,4) and (e_6,8) into the heap (the number in each pair represents the distance). Since e_5 has the smallest distance, it is removed from the heap and as there is currently no skyline points to compare, it is expanded into entries (e_2,7) and (e_1,9) which are also not dominated by any

skyline points and hence are inserted into the heap. Next, (e_2,7) is removed from the heap and expanded into leaf entries (d,7) and (e,11) which are inserted into the heap. Now, the next entry removed from the heap will be (d,7). Since it is a data point, it is guaranteed to be a skyline point and can be output. Subsequently, it is inserted into a separate list S for checking against future entries. The process then repeats and continues until the heap is empty. The answers for this example are points a, d, f.

We shall now briefly described two variants of the BBS algorithm. The first is the divide and conquer skyline (DSCSkyline) algorithm [75] and the second is the depth first skyline (DFS) algorithm [76]. Both work propose the checking of dominance between intermediate entries to improve the pruning power. Recall that BBS only checks the dominance between a skyline point and intermediate entries/data points, but not between entries. Formally, let the MBR of an entry e in a d-dimensional space be given by $[x_1, x_2, \ldots, x_d]$,$[y_1, y_2, \ldots, y_d]$ (the lower-left and upper-right corner points respectively). Consider the d points: $p_1 = (x_1, y_2, y_3, \ldots, y_d)$, $p_2 = (y_1, x_2, y_3, \ldots, y_d)$, ..., $p_d = (y_1, y_2, \ldots, x_d)$. The *dominance region* of e, $\mathrm{DR}(e) = \mathrm{DR}(p_1) \cup \mathrm{DR}(p_2) \cup \ldots \cup \mathrm{DR}(P_d)$. Any entry, g, that falls within $\mathrm{DR}(e)$ i.e. g is dominated by at least one of $\{p_1, \ldots, p_d\}$, is dominated by e. This is illustrated in Figure 2.4(a) for a 2-dimensional space. Entries e_2 and e_3 are dominated by e_1 because they are dominated by p_1 and p_2 respectively.



(a) Entry dominance. (b) DCSkyline principle.

Figure 2.4: BBS variants.

The first algorithm, DCSkyline, is only applicable for two dimensional spaces and is based on a divide and conquer paradigm. Similar to BBS, it starts by expanding the root node. From these entries, it finds the one with the shortest distance to the origin and uses its lower-left corner point to divide the data universe into *regions*. This is depicted in Figure 2.4(b) where the lower-left corner of entry e is used to divide the data space into three regions. Next, the algorithm is recursively applied to find the local skylines in regions R_1 and R_2, whose answers are guaranteed to be in the global skyline. This, however, does not hold for region R_3. Instead, the search for skyline points is recursively applied on region R'_3 which is the region in R_3 that is restricted by the right-most skyline point of R_1 and the left-most skyline point of R_2. This continues until all the skyline points are found.

The second algorithm, DFS, is based on the depth first search paradigm. The difference between DFS and BBS is the way DFS expands an intermediate entry. When an intermediate entry is expanded, all child entries are first checked against each other and any dominated entries removed. The remaining child entries are then inserted into the heap if it is not dominated by any skyline point (as in the original BBS algorithm) as well as any entry in the heap (a step which is specific to DFS). The rest of the steps are similar to BBS. [76] also describes how the memory usage can be reduced for two dimensional spaces when using the DFS approach.

***Distributed Skyline Computation* [4, 6]**

In [6], Balke et. al. focus on computing the skyline in a distributed environment (specifically, web information systems). They assume that each distributed web source associates a numerical score value to each *object* in the source and provides two basic ways of accessing its objects. For *sorted* access, objects are returned sorted by score. For *random* access, scores of independent objects can be retrieved in any order depending on which objects are desired.

For example, assume that scores can be associated with the food quality and service level of restaurants in a certain city. A person looking for a restaurant

might query two web sources where the first returns a list of restaurants sorted on food quality scores while the second returns a list of restaurants sorted on service level scores. In this scenario, a restaurant A dominates another restaurant B if the scores for food quality and service level of A is as good as B and either its food quality or service level is strict better than B. Such a query is analogous to the traditional skyline query by considering food quality and service level as attributes of restaurants (the respective scores are thus the attribute values) and a HIGHEST preference is specified on each of these attributes.

The authors adopt a similar approach as the B-tree approach. Each source is scanned using sorted access simultaneously until a match occurs. Then, random access is used subsequently for retrieving missing scores of objects that are partially seen. These retrieved objects are then compared pairwise and non-dominated objects returned. The authors also propose several heuristics to enhance their algorithm. A modified version of their algorithm which includes handling of categorical data is described in [4]. Finally, in [3], they adapt their algorithm to include answering of top-k queries (that has only a single objective function) to queries involving any number of objective functions (the extreme being a skyline query). However, their approach requires that all database objects be numerically scored.

Discussion on Existing Techniques for Skyline Computation

In terms of progressiveness, both BNL and DC require at least one scan of the dataset before some initial answers are returned. For SFS, if we factor in the sorting process, then initial answers will be delayed due to the sorting. Otherwise, answers can be returned progressively. Using B-trees, the initial answers are highly dependent on when the first match occurs. Even if the initial answers happen to be returned quickly, the algorithm might stall as it need to apply a main-memory skyline algorithm (typically BNL or DC) which is batch-based. This argument also applies to the distributed skyline computation due to its similarity with the B-tree technique. Finally, both NN and BBS (as well as its variants) can return initial

answers quickly and are progressive as well.

Although all existing algorithms support skyline queries involving HIGHEST and LOWEST preferences, their support for DIFF attributes vary. Assume that a DIFF attribute A_j is specified in a skyline query. If A_j has k distinct attribute values, then a skyline will have to be computed for each of the k values. A straightforward approach would be to group the data tuples into k groups based on their attribute values for A_j and then compute the skyline for each group. Such an approach, however, will prevent results from being returned progressively. A simple workaround would be to show the user the *partial skyline* of each group and then update the partial skylines progressively as evaluation continues. Note that it is important that the skyline points returned are not temporary i.e. false hits that will be subsequently replaced.

BNL, DC and SFS can support DIFF attributes without any problem. For the B-tree and distributed skyline computation techniques, it is not clear how DIFF attributes will be supported as this is not explicitly addressed in the original work. As mentioned in [69], NN cannot support DIFF attributes. For BBS (and its variants), an approach is proposed in [86]. However, it is important to note that as each entry in the multi-dimensional index can potentially contain data tuples of different distinct values, the pruning power of such approaches is greatly reduced. In the worst case, no pruning of intermediate entries is possible and the algorithm degenerates into a block-nested loop algorithm.

Finally, while most recent work focus on efficient algorithms for skyline computation, there are also several work closely related to skyline queries. In [42], Godfrey describes how to estimate the cardinality of skyline queries. In [43], the authors extend the semantics of skyline queries to make them more expressive. Finally, [62] examines the computation of a *thick skyline* which is similar to the traditional skyline except it includes neighboring points of a certain ϵ-distance.

Evaluating Pareto Preference Queries

As we mentioned in the first chapter, skyline queries and the more general pareto queries are instances of the maximum vector problem proposed by Kung et. al. [72]. Early work on solving the maximum vector problem typically assume that the data fit into the main memory. Algorithms devised include divide and conquer paradigm [72], parallel algorithms [91, 96] and those that are specifically designed to target at 2 or very large number of dimensions [79]. However, none of these algorithms deal explicitly with the case where the dataset cannot fit into the main memory. In fact, most of them are not applicable in such situations [9].

Since pareto queries are a more general form of skyline queries, it is natural to extend existing algorithms for skyline queries to pareto queries. However, it is important to note that not all skyline evaluation strategies can be extended to answer pareto queries. This is because some of the existing skyline algorithms assume that the underlying domain is ordered. In contrast, pareto queries may consist of preferences specified on **unordered** domains as well.

We shall now consider the applicability of existing skyline algorithms. First, the block nested loop algorithm is clearly applicable due to the generality of its operational semantics. For the divide and conquer algorithm, it is not suitable because it is not possible to find good medians for the m-way partitioning. For the sort-filter-skyline algorithm, it is also not suitable because it is not possible to do sorting on the unordered domains. It is possible to extend the B-trees technique and we will address this in chapter 5. For NN and BBS (including its variants), they are not applicable because they rely on spatial indexes which are not suitable for handling preferences specified on unordered attributes e.g. it does not make sense to have a spatial relationship between different color values.

Another approach to evaluating pareto queries which is adopted in Preference SQL [68] is to translate the queries into standard SQL and submit them to the relational database system for evaluation. This approach, however, is not an ideal one because the extensiveness of the various preferences makes it difficult to integrate

preference mechanisms within a relational engine. Moreover, this might result in a poorer performance. For example, in [9], the authors show experimentally that a skyline query that is translated into a nested SQL query takes longer to evaluate than those approaches that extend the capability of the relational engine directly. To this end, recent work [51] are starting to focus on optimizing preference queries algebraically through a tighter integration of the preference query optimizer and the SQL optimizer. In most cases, the underlying evaluation algorithm for pareto preferences is just a block nested loop algorithm. Clearly, more efficient algorithms are needed for evaluating pareto queries.

In [100], the authors propose a *Best* operator that can be used to answer pareto preference queries (as well as other complex preferences). The basic idea is to organize the search space such that the number of comparisons between tuples is minimized. Contrary to BNL, the *Best* operator does not require the preference relation to be a strict partial order. We shall now describe the *Best* operator.

The evaluation process is divided into a number of phases where each phase, i, makes multiple scans over a set of *candidate tuples*, C_i, producing a set of answers, Out_i. During the scan, tuples that are dominated by some tuple t are stored in a set D_t. In the first phase, the set of candidate tuples, C_1, consists of *all* tuples in the relation. The first tuple, t, is designated as the *selected* tuple and is compared with the rest of the tuples in the relation. Tuples dominated by t are placed in D_t while those that neither dominate t nor are dominated by t are placed in an *unresolved* set U_1. However, if a tuple s dominates t, t will be placed in the set D_s and the process continues with s as the selected tuple. At the end of the first scan, the selected tuple is compared with tuples in U_1 that it has not yet been compared with to eliminate any tuples it dominates. The selected tuple is then output as the answer (and stored in Out_1). The tuples in U_1 then become the candidate set and the procedure repeats in a similar manner until $U_1 = \emptyset$ which also ends the first phase. In each subsequent phase, i, the procedure is similar except that the candidate set is made up of tuples from the sets D_t, for each $t \in Out_{i-1}$. We note

that since we only consider preference relation of a strict partial order in our work, executing the first phase of the algorithm for *Best* is sufficient. Subsequent phases are necessary only if the preference relation does not follow a strict partial order.

A special data structure, called the β-tree [99], is built while executing the basic algorithm. In fact, the authors propose that the β-tree be pre-built and stored so that it can be used to efficiently answer queries of the same preference relation. Management issues of the β-tree is addressed in [101]. While the β-tree can be a useful index for preference queries, there could be many preference relations, requiring many β-trees to be built and stored, which may not be feasible. Moreover, the basic algorithm is not disk-based and there is no reported evaluation of the algorithm to show that it is effective, especially for large datasets. Nonetheless, the algorithm clearly requires at least one scan through the relation, making it unattractive for producing fast initial response time.

2.2.2 Quantitative Approach

Theoretical Frameworks and Models [2]

In [2], Agrawal and Wimmers present a framework for expressing and combining preferences in the quantitative approach. The paper essentially lays the theoretical foundations for a preference framework. In the framework, user preferences are specified using preference functions that map data tuples to scores. The authors propose a generic combine operation (which has a closure property) for combining preferences and illustrated the flexibility of their framework using real-life applications. However, they do not provide any declarative semantics of their preference queries in the paper.

In [71], a framework that integrates personalization and database queries using structured user profiles is proposed for database systems. The authors propose a preference model that relies on scores and numerical ranking. Given a standard query, the system first extracts the top k preferences from the profile of the user. Then, the preferences are integrated into the query to produce a new personal-

ized query that will produce answers that satisfy m top preferences and at least l preferences from the remaining $k - m$ ones.

The PREFER System [56] and the Onion Technique [20]

The work that explicitly deals with algorithmic issues associated with the implementation of features of the framework in [2] is [56]. The authors propose an algorithm to evaluate preference queries expressed as linear sums of the attribute values. They also develop a system called PREFER on top of a commercial database system to demonstrate the practicality of their approach. This approach is further extended in [58] where they develop a meta-broker system called MERGE for merging ranked results from multiple data sources [46]. In the PREFER system, a weighted preference function is applied directly on the attributes and is given by $a_1A_1 + a_2A_2 + \ldots + a_jA_j$ where $A_1, \ldots, A_j$ are attributes of a relation and $a_1, \ldots, a_j$ are the weights the user assigns to the attributes (other types of preference functions are addressed in [57]). In general, a higher weight for an attribute implies that the user "values more" about that attribute.

For efficient processing of preference queries, a set of materialized views are produced based on some predetermined preference vectors. In this way, answers to queries with similar preferences can be delivered quickly. The idea is to compute the smallest prefix of the view that has to be read to find the top tuple that satisfies the query; this process is repeated to retrieve the next top scoring tuple, and so on. For the scheme to perform well, sufficient views must be materialized. This incurs additional storage overhead, and raises the problem of picking the optimal views. Moreover, it is effective only if the users are expected to share similar preference functions. Furthermore, there is no guarantee of correctness as long as the view corresponding to the user preference vectors is not materialized, i.e., in this case, it can only provide approximate answers.

The closest work to PREFER is by Chang et. al. [20]. They propose an indexing technique called the "Onion" technique to speed up evaluation of linear

optimization queries. The scheme precomputes a set of convex hulls of the tuples: the first (outermost) convex hull is computed on the entire dataset, the second convex hull is on the remaining data, and so on. Answers can be obtained progressively by scanning the convex hulls beginning from the first convex hull until such numbers of tuples as specified by the users. This is possible because the first answer is definitely in the first convex hull. While the technique can provide first answers quickly, each convex hull also contains many tuples that are not in the answer set. Thus, the cost to scan the convex hulls one at a time can increase the overall processing cost. Comparison between the Onion technique and PREFER is discussed extensively in [58] where PREFER is also shown to outperform the Onion technique.

It is interesting to note that the convex hull is a subset of the skyline. While the convex hull contains only points that may be optimal for a *linear* scoring function, the skyline contains all points that may be optimal for *any* monotonic scoring functions. The computation of the convex hull has been studied extensively [89] but most of these algorithms require that the dataset be resident in main memory. Recently, [8] proposes a branch and bound algorithm for finding the convex hull of large datasets indexed by R-trees.

Evaluating Top-k Queries

Evaluation of top-k queries has been studied extensively in the last decade and there is a large body of work in this area. We shall classify these work generally into three categories, according to the context the work originally assume.

Top-k Queries over Multiple Data Sources

This category addresses the evaluation of ranked queries when the information about each *object* of interest is distributed across multiple data sources. Systems that evaluate such queries are typically middleware systems that are built on top of other subsystems and integrate results from them. These subsystems usually

contain heterogeneous data e.g. one could be a text repository while another is a multimedia repository. The data in these subsystems are also inherently fuzzy. As a result, these subsystems return *graded* sets consisting of pairs (x,g) where x represents the object and g is the grade (real number in [0,1]) that measures how closely match x is to the subquery. To answer a compound query, an *aggregation function* (or *scoring function*) is adopted to combine the respective scores to obtain an overall score. The k top scoring objects are then returned.

There are two ways of accessing objects in a subsystem. The first is *sorted access* where the middleware system asks the subsystem to return a list of objects sorted by their grades. These objects can be returned in batches e.g. in tens. The second is *random access* where the middleware system requests for the grade of a specific object from the subsystem. Performance of the algorithms is typically measured in terms of the total number of objects obtained from the various subsystems under sorted and random access.

For ease of discussion of the algorithms, we shall assume that the database has N objects each with attributes $A_1, \ldots, A_j$. Each object, R, has attribute values $a_1, \ldots, a_j$ for attributes $A_1, \ldots, A_j$ respectively where $a_i \in [0,1]$ for $1 \leq i \leq j$. We shall think of this database as consisting of j sorted lists, $L_1, \ldots, L_j$. Each sorted list, L_i, consists of N entries each of the form (R,a_i). Each list is sorted in descending order of the a_i values.

One of the first algorithm in the literature is Fagin's algorithm (FA) [36, 37]. It works as follows. First, the sorted lists are accessed in parallel using sorted access i.e. the first entry of each list is picked followed by the second entry and so on, until there are at least k "matches". A match represents an object that has been seen in each of the j lists. Second, for each seen object, R, find the value of its ith attribute by doing a random access on list L_i. Of course, we need not do this for the matches since we already know all their attribute values. Finally, apply the aggregation function $f(R) = f(a_1, \ldots, a_j)$ on each object R that has been seen to compute its score. The highest scoring k objects are subsequently output (with

ties broken arbitrary). A similar approach to FA is proposed by Chaudhuri and Gravano [21] using "filter conditions". For example, it is possible to specify a query to a subsystem asking for objects with grade 0.9 or higher in [21].

However, there are situations where FA performs poorly. In response to this, Fagin et. al. propose the threshold algorithm (TA) in [38, 39]. It works as follows. First, do sorted access of each of the sorted list in parallel. As each object R is seen in sorted access from any of the lists, retrieve the rest of the attribute values of R by doing random access on the other lists. Then, compute the score of R. R is kept, together with its score, if its score is one of the k highest seen so far (ties are broken arbitrary to ensure that only k objects and their scores are kept in memory at any time). For each list, let $\underline{x_i}$ be the attribute value of the last object seen in L_i under sorted access. A *threshold value* is then computed by applying the aggregation function on $(\underline{x_1}, \ldots, \underline{x_j})$. Once there are at least k objects whose scores are at least equal to the threshold value, the algorithm terminates and the k highest scoring objects are output. Nepal and Ramakrishna [82] independently discover an algorithm similar to TA. They focus mainly on the min aggregation function in their work. Moreover, their notion of optimality is weaker than TA.

In [48], the authors propose an enhanced version of TA called Quick Combine (QC) by using heuristics that can potentially speed up the evaluation. The algorithm works as follows. Initially, QC retrieves the first m entries from each list and computes an *indicator value*, $\triangle_i$, for list i. If x_i is the attribute value of the last object retrieved from L_i and y_i is the attribute value of the m^{th} object retrieved prior to x_i, then $\triangle_i$ is given by $w_i \cdot (y_i - x_i)$ where w_i is an optional weight given by the user for L_i to indicate its relative importance compared to the other lists. Thus, $\triangle_i$ indicates the rate of change of values for L_i and is used to control which list to retrieve values from during query processing. Subsequent steps are similar to TA except when deciding which list to pick the next entry from in sorted access, the one with the larger $\triangle_i$ (ties broken arbitrary) will be accessed first. $\triangle_i$ is also continuously updated during evaluation.

There are also several variations of the above algorithms which are designed to handle specific restrictions that may be imposed by the environment. The first scenario is when sorted access is restricted. For example, several web repositories in the Internet only support random access. Work in this area include [12, 39, 78]. The basic idea behind their approaches is that when calculating the threshold value, for those lists where sorted access is restricted, $\underline{x_i}$ is assigned a value of 1. The second scenario is when random access is restricted. For example, the subsystems may be search engines which always return sorted lists of results. Work in this area include [39, 49, 104].

It is interesting to note that although the above algorithms are targeted at an environment with multiple data sources (possibly distributed), they can be adapted to evaluate numerical preference queries in a centralized database system. In fact, we can view all the data as residing in a single relation with one column representing the object id and the other columns are the attributes of the object. The lists can be further viewed as indexes e.g. B^+-trees which enable entries to be accessed in sorted order of descending attribute values and also in random order by making direct searches through the indexes. Moreover, as discussed in [40, 41], it is possible to modify the aggregation function to incorporate user preferences.

Top-k Queries in Relational Systems

Traditionally, top-k queries over relational database systems are evaluated using a cursor-based approach. The entire query is submitted to the database system and then the top k results are extracted through a cursor. However, this can result in long response times as well as redundant work done by the database engine. In [13, 14], Carey et. al. propose extending SQL with a `STOP AFTER` clause to support top-k queries and they describe how traditional DBMS can be extended to provide integrated support for such queries, which can lead to orders-of-magnitude improvements in many cases. A new operator, the *Stop* operator, is proposed for encapsulating the function of the `STOP AFTER` clause. Additional strategies for

processing `STOP AFTER` queries are described in [15] and [31]. It is important to note that scoring in such queries is done through the traditional SQL `ORDER BY` clause. Consequently, if the scoring function is not based on the column values, then at least one sequential scan of the database is required.

[11, 22] study the evaluation of top-k queries over relational databases for a wide variety of scoring functions. They call such queries *top-k selection queries*. A top-k selection query consists of a set of target values and the database system uses some scoring function e.g. Euclidean distance to decide how closely each tuple in the database matches with the target values in the query. The basic idea is to translate a top-k query into a range query that can be evaluated efficiently by the system. The difficulty here is how to determine the right range query. To this end, the authors make use of statistics available to the DBMS to map the top-k query into a suitable range that will encapsulate the k best matches. A similar approach but using sampling is proposed in [24]. The query model used in [24] is also slightly different as the focus is on returning *approximate* answers to the query.

While a top-k selection query selects ranked tuples from a single relation, a *top-k join query* selects ranked tuples from the results of a join operation. [81] discusses the general problem of doing top-k joins while [59, 60] propose a progressive algorithm for evaluating top-k join queries in relational databases.

Finally, it is important to note the difference between a top-k selection query and a numerical preference query. In the top-k selection query, target values are specified and scores are computed with respect to the target values. On the other hand, a numerical preference query returns the top k answers that maximize a given linear function over the relation's attributes.

Top-k Queries in Other Contexts

Finally, we shall briefly examine the application of top-k queries in other contexts. In [19], the authors address the issue of supporting *expensive predicates* for top-k queries. For example, a user can specify his/her preference for attribute `size` by

fuzzy predicates such as *large*. An algorithm called *MPro* is proposed to evaluate such queries with minimal number of "probes". [5] proposes an algorithm for mobile environments which have strict real-time constraints. [106] examines the issue of making a top-k view runtime self-maintainable with high probability. [23, 77] look at optimizing top-k queries on multimedia and web repositories while [16] addresses the issue of query refinement of top-k queries.

2.2.3 Other Approaches

We shall now briefly cover some approaches that are generally not classified as quantitative or qualitative but do support user preferences in some way.

Nearest Neighbor Searches

Given a multi-dimensional point p and a distance metric, a nearest neighbor search returns points closest to p, in ascending order of their distances. To date, there is a large body of work on evaluating nearest neighbor queries, particularly, on finding the k Nearest Neighbors (KNN) of some focal point (see e.g. [7, 102] and references therein). Existing database algorithms typically employ a R-tree (or some other partitioning strategies) for indexing the data and adopt a branch and bound approach for evaluating nearest neighbor queries. In [93], a *depth-first* algorithm is proposed. It starts from the root node and recursively visits nodes nearest to the query point. Nodes further than existing nearest neighbors are pruned. In [54, 55], *best-first* algorithms are proposed. They store visited nodes in a heap and follow those closest to the query point.

Nearest neighbor search is not an effective mechanism for supporting user preferences *directly*. For example, a nearest neighbor search for an *ideal* house that costs \$0 and distance 0km from the beach would certainly return some interesting answers but would neglect those that are extremely cheap but further from the beach. Furthermore, non-interesting houses that are already dominated by others may be returned. However, nearest neighbor searches can be invaluable as a mech-

anism to *support* evaluation of preference queries. This is exemplified by existing skyline algorithms that utilize nearest neighbors for partitioning and pruning.

Optimization Under Parametric Aggregation Constraints

A recent work [47] addresses pareto queries. The authors study a new class of queries called OPAC (Optimization under Parametric Aggregation Constraints) queries that retrieve tuples based on the constraints specified on the tuples' attributes. For example, a project leader might request for a set of developers having the smallest total salaries with total number of years of experience no less than ten. However, the context of their work is not directly related to ours because our focus is on individual attribute values while [47] considers attribute values *collectively*.

Cooperative Query Answering

While preference queries allow users to specify his/her preferences up front, cooperative query answering (CQA) [63, 64] allows a user to specify his/her query as usual and remedy the case when the query is unsuccessful. Some techniques provide additional information to the user to help him/her crafts follow-up queries. Other techniques create the follow-up queries automatically by augmenting the original queries and resubmitting them to the system again. In either case, these CQA approaches provide limited means by which the user can specify his/her preferences with regard to specific components of the query. An example of a cooperative information system is CoBase [28] which allows adding of preferences realized as conditions to the query. If no exact matching answers are found for the query, the system *relaxes* the conditions till some approximate answers are found. A broader survey of CQA can be found in [80].

Preferences in Artificial Intelligence

Preferences has also been extensively studied in artificial intelligence. For example, preference reasoning [10, 98, 103] focuses on developing mechanisms for making

inferences about preferences and solving configuration or decision problems. Intelligent databases seek to embed preferences and priorities into the semantics. A central notion in these work is the *ceteris paribus* [52] preference: preferring one situation/solution/tuple to another, *all else being equal*. As with the logical approach to preferences, these work can potentially be applied to the relational data model but a detailed study for adapting these approaches remains still to be done.

CHAPTER 3

Progressive Skyline Computation

In this chapter, we shall focus on the progressive computation of the skyline. Recall that the skyline of a set of points S are those points in S that are not dominated by any other points. In a skyline query, users specify whether they prefer high, low or different values of various attributes. While the skyline has been studied previously as the maximum vector problem [72], adapting existing algorithms proposed for the maximum vector problem to the relational database context is challenging because most do not handle situations where the dataset does not fit into main memory. Moreover, most existing algorithms that evaluate skyline queries in the database context are batch-oriented. We further note that the domains of the attributes in a skyline query must have a natural total ordering e.g. integers and floats. In the next chapter, we consider the case where the domains may be *partially ordered*.

This chapter is organized as follows. First, we shall introduce the skyline operator as originally proposed in [9]. We shall adopt terminology similar to [9] so that our discussion is consistent with existing work. Then, in the following two sections, we present two novel indexing algorithms to compute the skyline of a set of points progressively. Section 3.3 presents our extensive experimental study and the last section provides a summary for this chapter.

3.1 The Skyline Operator

To the best of our knowledge, the first work that addresses skyline queries in the context of databases is [9] where Borzsonyi et. al. extended the SQL's `SELECT` statement by an optional `SKYLINE OF` clause. As described in [9], the `SKYLINE OF` clause is evaluated after the `SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ...` part of the query, but before the `ORDER BY` clause (and `STOP AFTER` if supported). The `SKYLINE OF` clause selects points that are not dominated by other points. Formally, a point $p = (p_1, \ldots, p_k, p_{k+1}, \ldots, p_l, p_{l+1}, \ldots, p_m, p_{m+1}, \ldots, p_n)$ dominates another point $q = (q_1, \ldots, q_k, q_{k+1}, \ldots, q_l, q_{l+1}, \ldots, q_m, q_{m+1}, \ldots, q_n)$ for the query:

`SKYLINE OF` d_1, `MIN`, ..., d_k `MIN`, d_{k+1} `MAX`, ..., d_l `MAX`, d_{l+1} `DIFF`, ..., d_m `DIFF`

if the following three conditions hold:

$$\begin{aligned} p_i &\leq q_i \quad \text{for all } i = 1, \ldots, k \\ p_i &\geq q_i \quad \text{for all } i = k+1, \ldots, l \\ p_i &= q_i \quad \text{for all } i = l+1, \ldots, m \end{aligned}$$

In the `SKYLINE OF` clause, the `MIN` and `MAX` *annotations* mean that the corresponding dimensions should be minimized and maximized respectively, and the `DIFF` annotation denotes that two points with different values for the same dimension may both be part of the skyline i.e. that dimension is ignored in the comparison if they have the same value. For ease of presentation, we have put the `MIN` dimensions first and `DIFF` dimensions last although it does not make a difference how they are arranged. Note that if $p_i = q_i$ for all $i = 1, \ldots, m$, then p and q are incomparable and both are part of the skyline.

As an example, consider a tourist who is looking for a hotel and wishes that both price and distance be minimized. The query can be written in SQL as follows:

```
SELECT *
FROM hotels
WHERE rating > 2
SKYLINE OF price MIN, distance MIN;
```

where `hotels` (`hotelid`, `address`, `typeOfRoom`, `rating`, `price`, `distance`) is a

relation on hotel information. `rating` represents the rating of the hotel (here, we assume its type is integer, and the value represents the number of stars). `price` captures the room rates, and `distance` indicates the distance from the hotel to the city. Should the user want to maximize the value of an attribute (e.g., one may want to maximize the rating), then the `MAX` annotation can be used. Similarly, the user may use the `DIFF` annotation to indicate that two hotels with different room types are acceptable.

3.2 Progressive Skyline Computation Algorithms

In this section, we propose our two novel indexing methods to compute the skyline progressively. For pedagogical reasons, we shall assume that the database, D, contains $|D|$ d-dimensional points. Moreover, we assume that the skyline operation involves all the d dimensions, and that dimension i has k_i distinct values, $1 \leq i \leq d$. Let a_{ij} denote the jth distinct value of the ith dimension. Without loss of generality, we assume that $a_{i1} > a_{i2} > \ldots > a_{ik_i}$. In addition, we assume that the domains for all dimensions are the same and are totally ordered. In presenting the proposed schemes, we shall also restrict our discussion to the `MAX` annotation only. Towards the end of this section, we shall discuss how the schemes can be easily generalized to handle skyline queries involving fewer than d dimensions, other annotations (i.e., `MIN` and `DIFF`), as well as other related issues.

3.2.1 Bitmap: A Bitmap-based Algorithm

To support progressive skyline computation, for each point examined, the Bitmap scheme asks the question: "Is this point dominated by another point?". The point is an interesting point if the answer is negative, and we can return it immediately. As such, the method is completely non-blocking, and the initial response time is short compared to existing schemes. Intuitively, to realize this, we need to examine all points in the database. We avoid this by exploiting a bitmap structure. Since

bitwise operations are fast, we can efficiently determine whether a point is an interesting point or not. We now describe how the bitmap structure is constructed.

Construction of the Bitmap Structure

Assume that the domain of all dimensions are integral values. Consider a point $x = (x_1, \ldots, x_d)$. We use k_i bits to represent x_i. k_i is the number of distinct values of dimension i. Let the jth bit of the k_i bits corresponds to a_{ij}. Note that since we are considering the `MAX` annotation, the first bit corresponds to a_{i1} (which corresponds to the largest value in the dimension), the second bit corresponds to a_{i2} (which corresponds to the second largest value), and so on. If x_i is the a_{iq}th distinct value of dimension i, then the k_i bits representing x_i are set as follows: bits 1 to $q-1$ are set to 0, and bits q to k_i are set to 1. By setting the respective k_i bits for each x_i value, x can be represented as a m-bit vector where $m = \sum_{i=1}^{d} k_i$.

For efficient computation, the array of vectors obtained from all points are transposed into an array of *bit-slices* and stored as slices in secondary storage in descending order of the dimensional values. However, we note that the bitmap structure can impose a large space as well as high processing overheads for non-discrete domains or domains with large cardinality. We shall address these issues towards the end of this section.

Example 3.1. Figure 3.1 shows an example. Here, we have a table containing four 3 dimensional data points (Figure 3.1(a)). The first dimension has 4 distinct values (4, 3, 2, 1), the second dimension has 3 distinct values (3, 2, 1) and the third dimension has 2 distinct values (2, 1). The corresponding bitmap structure is shown in Figure 3.1(b). Consider the second point `(3 2 1)`. The value in its first dimension is 3, which is the second largest value. So, only the bit corresponding to 4 is set to 0, while the rest are set to 1, resulting in the sequence `0111`. Similarly, for the second dimension, its value is 2, and so only the bits corresponding to values larger than 2 (in this case, only one of them which has value 3) will be set to 0, while the rest are set to 1. This leads to the sequence `011`. Finally, using the same

logic, the bit sequence corresponding to the third dimension is 01.

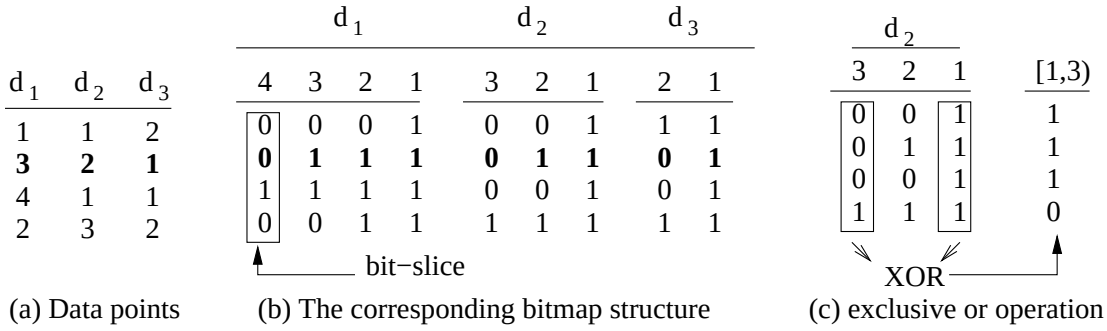


Figure 3.1: An example to illustrate the bitmap-based method.

In Figure 3.1(b), we also illustrate how a bit-slice looks like. Let us now examine an important property of these bit-slices.

Property 3.1. *Assume that dimension i has k_i distinct values, $a_{i1}, \ldots, a_{ik_i}$ where $a_{i1} > a_{i2} > \ldots > a_{ik_i}$. Let $BS_{a_{ij}}$ denotes the bit-slice for the jth distinct value, a_{ij}, of dimension i. Then, the 1s in $BS_{a_{ij}}$ represent those points having values greater or equal to a_{ij} for dimension i while the 1s of the* ***immediate preceding*** *bit-slice of $BS_{a_{ij}}$ i.e. the bit-slice for the $(j-1)$th distinct value (for $j > 1$), represent points having values strictly greater than a_{ij} for dimension i.*

For example, consider the bit-slice for second distinct value of the first dimension in Figure 3.1(b). From the bit-slice for value 3 of the first dimension, we know that the second and third points have values greater or equal to 3. On the other hand, its immediate preceding bit-slice tells us that only the third point has value strictly greater than 3. This brings us to an important theorem:

Theorem 3.1. *Given any two distinct bit-slices $BS_{a_{ij}}$ and $BS_{a_{ik}}$ where $a_{ij} > a_{ik}$, executing a bitwise exclusive or operation between them will result in a bit-slice whose 1s represent those points having values in the range of $[a_{ik}, a_{ij})$.*

Proof. WLOG, consider a point that is represented by the nth bit in the bit-slices. Assume that it has value a_{iq} for dimension i. First, if $a_{iq} < a_{ik}$, then bit n in bit-slices $BS_{a_{ij}}$ and $BS_{a_{ik}}$ will be set to 0. Hence, bit n will remain 0 in the resulting

*exclusive-or*ed bit-slice, indicating that a_{iq} is not in the range of $[a_{ik}, a_{ij})$. Second, if $a_{iq} \geq a_{ij}$, then bit n in bit-slices $BS_{a_{ij}}$ and $BS_{a_{ik}}$ will be set to 1. Hence, bit n will be set to 0 in the resultant bit-slice, indicating that a_{iq} is not in the range of $[a_{ik}, a_{ij})$. Finally, if $a_{ik} \leq a_{iq} < a_{ij}$, then bit n is set to 0 in bit-slice $BS_{a_{ij}}$ but set to 1 in bit-slice $BS_{a_{ik}}$. Hence, bit n will be set to 1 in the resultant bit-slice, indicating that a_{iq} is indeed in the range of $[a_{ik}, a_{ij})$. □

An example is illustrated in Figure 3.1(c). We execute a bitwise *exclusive or* operation on the bit-slices for values 3 and 1 of the second dimension, resulting in the bit-slice whose 1s indicate points with values in the range of [1, 3). As we will see later, this theorem is frequently applied to derive the bit-slices required for evaluation e.g. to derive the bit-slice whose 1s represent points having the same value for a specific dimension (as shown by the following).

Corollary 3.1. *Let $BS_{a_{ij}}$ be the immediate preceding bit-slice of $BS_{a_{ik}}$. Executing a bitwise exclusive or operation between these two bit-slices will result in a bit-slice whose 1s represent those points having only values a_{ik} for dimension i.*

In order to facilitate our subsequent discussion on query evaluation we shall now present a number of definitions. Note that d_i represents dimension i of the dataset for $1 \leq i \leq d$.

Definition 3.1. Let $BitSlice(a_{ij}, d_i)$ be the bit-slice for value a_{ij} in dimension i. The 1s in the bit-slice represent those points having values $\geq a_{ij}$ for dimension i.

Definition 3.2. Let $PreSlice(a_{ij}, d_i)$ be the bit-slice that *immediately precedes* value a_{ij} in dimension i. Intuitively, the 1s of this bit-slice represent points having values $> a_{ij}$ for dimension i. If there is no preceding bit-slice, $PreSlice(a_{ij}, d_i)$ is equal to 0.

For example, in Figure 3.1(b), $BitSlice(2, d_2)$ refers to the bit-slice of the second dimension where the 1s represent points having values ≥ 2 i.e. `0101`. On the same figure, $PreSlice(2, d_2)$ refers to the bit-slice of the second dimension where the 1s represent points having values > 2 i.e. `0001`.

Due to the unique way in which our bitmap is structured, only the bit-slice for the first distinct value, a_{i1}, will consist of 1s representing points having values equal to a_{i1}. In general, the bit-slice whose 1s represent points having a specific value for a particular dimension will have to be *derived* and is defined as follows.

Definition 3.3. Let $OrigSlice(a_{ij}, d_i)$ be the bit-slice which is derived by executing a bitwise *exclusive or* operation on $BitSlice(a_{ij}, d_i)$ and $PreSlice(a_{ij}, d_i)$. From Corollary 3.1, the 1s of this bit-slice represent points having values a_{ij}.

Evaluating Skyline Queries

We shall now describe how to use the bitmap structure to evaluate a skyline query. Figure 3.2 gives the algorithmic description of the proposed Bitmap technique. The algorithm starts by looping through each point x in the database. For each point in the database, $BitSlice(x_i, d_i)$ are retrieved from the bitmap structure for each dimensional value of x and bitwise *and* together (line 5). Similarly, $PreSlice(x_i, d_i)$ are retrieved from the bitmap structure for each dimensional value of x and bitwise *or* together (line 6). The two resultant bit-slices, S_A and S_B respectively, are then bitwise *and* together (line 7). The resultant bit-slice, S_C is then checked (lines 8-9). If it is zero, we can conclude that no points in the database dominates x i.e. x is a skyline point and we output x.

Algorithm Bitmap
1. **foreach** point $x = (x_1, x_2, \ldots, x_d)$ in the database
2. $\quad S_A \leftarrow 1$ // set each bit of S_A to 1
3. $\quad S_B \leftarrow 0$
4. $\quad$ **for** $i = 1$ to d
5. $\quad\quad S_A \leftarrow S_A \;\&\; BitSlice(x_i, d_i)$
6. $\quad\quad S_B \leftarrow S_B \mid PreSlice(x_i, d_i)$
7. $\quad S_C \leftarrow S_A \;\&\; S_B$
8. $\quad$ **if** $S_C == 0$
9. $\quad\quad$ output x

Figure 3.2: Bitmap-based skyline computation algorithm.

The following result demonstrates that given any skyline query, the Bitmap algorithm correctly derives the bit-slice, S_C, which indicates whether any points in the dataset dominates the candidate point x.

Theorem 3.2. *Consider a d-dimensional dataset. Assume a skyline query Q specifying the* `MAX` *annotation on each dimension d_i, for $1 \leq i \leq d$. For a candidate point $x = (x_1, x_2, \ldots, x_d)$, the Bitmap algorithm correctly computes the bit-slice S_C whose 1s indicate only points that dominate x.*

Proof. Let us first examine the properties of bit-slices S_A and S_B:

$$S_A = BitSlice(x_1, d_1)\ \&\ BitSlice(x_2, d_2)\ \&\ \ldots\ \&\ BitSlice(x_d, d_d)$$

where & represents the bitwise *and* operation. Thus, S_A has the property that the nth bit is set to 1 iff each dimensional value of the nth point has value greater or equal to the value of the corresponding dimension in x. In other words, the 1s in S_A represent points having dimensional values strictly better or equal to x.

$$S_B = PreSlice(x_1, d_1)\ |\ PreSlice(x_2, d_2)\ |\ \ldots\ |\ PreSlice(x_d, d_d)$$

where | represents the bitwise *or* operation. Thus, S_B has the property that the nth bit is set to 1 iff the nth point has some of its dimensions' values strictly greater than the value of the corresponding dimension in x. In other words, the 1s in S_B represent those points whose dimensional values are strictly greater than x for at least one of their dimensions. Next, we examine the property of bit-slice S_C which is derived from bit-slice S_A and S_B:

$$S_C = S_A\ \&\ S_B$$

Thus, S_C has the property that the nth bit is set to 1 iff each dimensional value of the nth point has value greater than or equal to the corresponding dimension's value in x **and** some of its dimensions' values are strictly greater than the corresponding dimension's value in x. Hence, if S_C is not zero, we can conclude that there exists a point that is as good or better than x in all dimensions and better than x in at least one dimension i.e. x is dominated by some points. Conversely, if the resultant bit-slice is zero, it tells us that there is NO such point in the database that dominates x. Thus, the 1s in S_C can only represent points that dominate x. □

Example 3.2. Referring to our example in Figure 3.1. we now illustrate how to determine whether the second point `(3, 2, 1)` is in the skyline or not. First, we derive bit-slice S_A:

$$S_A = \texttt{0110} \ \& \ \texttt{0101} \ \& \ \texttt{1111} = \texttt{0100}$$

Next, we derive bit-slice S_B:

$$S_B = \texttt{0010} \mid \texttt{0001} \mid \texttt{1001} = \texttt{1011}$$

Finally, we carry out the bitwise *and* operation of S_A and S_B:

$$S_C = \texttt{0100} \ \& \ \texttt{1011} = \texttt{0000}$$

Since the answer is zero, no points in the database dominates `(3, 2, 1)`. Thus, it is in the skyline. This example clearly shows that as each point is examined, we can determine easily whether it is in the skyline!

Before we discuss about efficiency issues of the Bitmap algorithm, let us consider the following result.

Theorem 3.3. *Let D be the set of points in the database. Suppose there are d dimensions, and dimension i has k_i distinct values, $1 \leq i \leq d$. Moreover, let a_{ij} denotes the jth distinct value of the ith dimension. WLOG, we assume that $a_{i1} > a_{i2} > \ldots > a_{ik_i}$. Given a candidate point $x = (x_1, x_2, \ldots, x_d)$, the Bitmap algorithm needs to examine at most 2d bit-slices to determine whether x is in the skyline or not. In addition, there are a total of at most 2d - 1 bitwise operations (i.e., d bitwise and and (d - 1) bitwise or operations).*

Proof. In the Bitmap algorithm, we need to compute the bit-slices S_A, S_B and S_C. Bit-slice S_A is the resultant bit-slice from the bitwise *and* operations on $BitSlice(x_1, d_1), BitSlice(x_2, d_2), \ldots, BitSlice(x_d, d_d)$. Hence, it needs to examine d slices and there are $d-1$ *and* operations. Bit-slice S_B is the resultant bit-slice from the bitwise *or* operations on $PreSlice(x_1, d_1), PreSlice(x_2, d_2), \ldots, PreSlice(x_d, d_d)$. Hence, it needs to examine d slices and there are $d-1$ *or* operations. For bit-slice S_C, it is the result of executing a bitwise *and* operation on S_A and S_B. Therefore, the algorithm needs to examine a total of at most $2d$ bit-slices and there are a total of at most $d-1+d-1+1$ i.e. $2d-1$ bitwise operations. □

Although the number of bit-slices that need to be examined and operated on is proportional to the number of dimensions involved in the skyline query (which is typically less than 10), the size of the bit-slice itself can have an impact on performance. The size of each bit-slice depends on the number of points. A large dataset will result in large bit-slices even though we are using only one binary bit per point. As an example, our experimental datasets typically consist of 100000 points. These require 100000 bits each i.e., 12.5 KB of storage for each bit-slice. Consequently, during query evaluation, a high I/O as well as CPU cost is incurred to retrieve the bit-slices and execute the bitwise operations on them. To this end, we propose the following two optimizations to make the algorithm more efficient.

Bit-slice Segmentation

We propose that each bit-slice be segmented in a manner similar to that discussed in [83, 92]. The general idea is to break the bit-slice for a value x_i into segments such that each segment can fit into a disk page. The optimal size of each segment is application specific and has to be determined experimentally (which is what we have done for our experimental studies). Next, for each segmented bit-slice, we create a bit-slice index entry to reference the various segments. Figure 3.3 illustrates a typical bit-slice index entry for the value x_i.

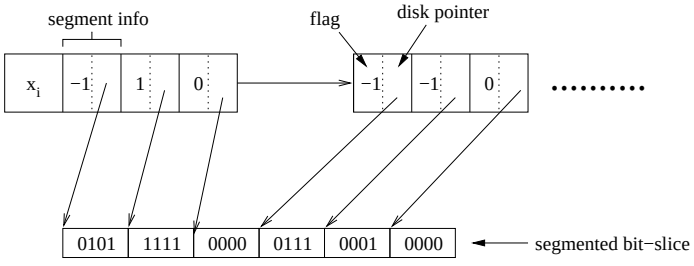


Figure 3.3: A bit-slice index entry.

Each index entry resides in a B-tree leaf page and may be followed by successive leaf pages if the size of the entry is too large to fit into a single leaf node. Each entry consists of a set of *segment information blocks* each containing two attributes. The first is an *indicator flag* while the second is a *disk pointer* to a segment of the

bit-slice on disk. The flag can take on three possible values. A value of 1 indicates that the segment has all its bits set to 1 while 0 indicates that all its bits are set to 0. A value of -1 indicates a mixture of 1s and 0s in the segment. We can represent the flag using two binary bits. Assuming a typical disk pointer of 8 bytes i.e. 64 bits, the size of each segment information block is thus 66 bits. Given the size of each segment information block, we can then calculate the extra storage incurred.

Now, instead of retrieving and processing the whole bit-slice in one shot, we now process segments of each required bit-slice in an incremental fashion. The advantage here is that it is now not necessary to examine all the segments of the bit-slice. For example, after applying the bitmap algorithm on the first segment of each required bit-slice, if the resulting bit-slice is not zero, we can immediately conclude that the current point is dominated by some other points without examining the rest of the segments. Furthermore, if we encounter a segment whose flag is set to 1 or 0, we do not even need to retrieve the segment, thus reducing a substantial amount of I/O cost.

As an example, Figure 3.4 shows the bit-slices and the index entries for the first dimension of Figure 3.1(b). Figure 3.4(a) shows the case when no segmentation is applied to the bit-slices while Figure 3.4(b) shows the case when segmentation is applied. Notice that with segmentation, there is one disk pointer for each segment. However, not every segment needs to be accessed. For example, since the last index entry has both its flags set to 1, there is no need to retrieve the segments as the flags indicate that they all have their bits set to 1, thereby saving a few I/Os. Hence, we believe that with segmentation, the performance of our bitmap algorithm can improve substantially and we adopt it in our implementation.

Skyline Cache

We propose another simple enhancement to our Bitmap algorithm by maintaining a list of skyline points found during processing. Subsequently, before the bitmap structure is consulted, a point is checked against this list of cached skyline points

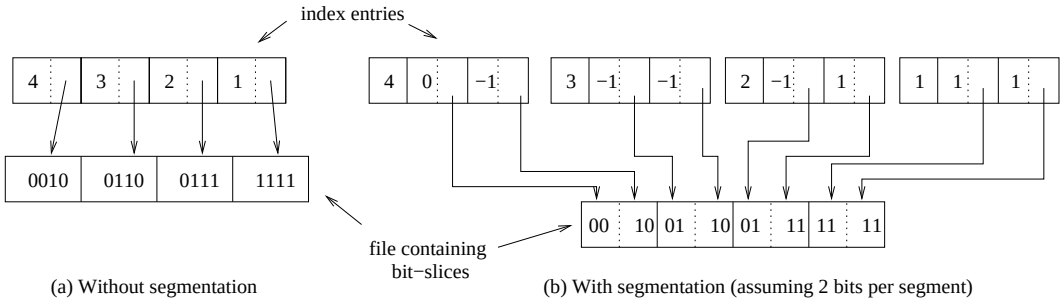


Figure 3.4: An example to illustrate bit-slice segmentation.

first. If any of the cached skyline points dominates it, the point will be eliminated. Otherwise, the normal Bitmap algorithm is applied to determine whether any points dominates the point under consideration. Caching of skyline points is particularly useful in situations where the number of skyline points is small as most of them can fit in the cache. This will further lead to the elimination of a large number of non-skyline points without consulting the bitmap structure, thus saving a significant amount of time.

Before we end this subsection, we would like to comment on the space efficiency of our bitmap structure. As we mentioned earlier, a large dataset will result in large bit-slices that will take up a substantial amount of secondary storage. Furthermore, if the domains are non-discrete or have large cardinalities, the space requirement is even higher. For non-discrete domains, we propose that their values be converted to discrete values whenever possible. For example, if an attribute is the price of some commodity in dollars and cents, the values stored in the bitmap structure will be the total number of cents instead. However, if the number of digits after the decimal point is large, it may be impractical to use the bitmap structure at all.

For sparse bit-slices, a possible solution is to use compression. For example, run-length encoding can be used to reduce the number of bits stored or in some cases, we can store the actual ids rather than using bits to conserve space. We believe that compression along the lines of [105] will be the most helpful in our work. However, it is important to note that our bitmap structure is augmented with additional information to speed up query evaluation and thus differs from existing implementations. Most notably, the density of the bit-slices increases as

the values get smaller, ending with a bit-slice consisting of only 1s for the smallest value. Since our main focus in this dissertation is on the efficiency aspects of those structures and algorithms suitable for evaluating skyline queries, addressing the issue of space overhead (which is an important topic on its own) is beyond the scope of this dissertation.

3.2.2 Index: A B$^+$-tree-based Algorithm

The Index scheme exploits a transformation mechanism that maps high dimensional points into single dimensional space and a B$^+$-tree structure [29] to index the transformed points. It assumes that the domains for all dimensions are in the range [0,1]. Towards the end of the section, we shall discuss how to handle databases whose dimensions' domains are different. The scheme works as follows. Let $x = (x_1, x_2, \ldots, x_d)$ be an arbitrary point. Let x_{max} be the largest value among all the d dimensions of the data point. Let the corresponding dimension for x_{max} be d_{max}. The data point is mapped to y over a single dimensional space as follows

$$y = d_{max} + x_{max} \tag{3.1}$$

We note that the transformation actually organizes the data space into different partitions based on the dimension which has the largest value, and provides an ordering within each partition. After the transformation, any single dimensional indexing structure can be used to index the transformed values. We adopt the B$^+$-tree structure in our implementation. However, we assume that the B$^+$-tree leaf nodes are linked to both the left and right siblings [90]. Moreover, we assume that the high dimensional points are kept at the leaf nodes of the tree.

We note that the transformation in Equation 3.1 is a special case of the more general transformation function used in iMinMax(θ) [84]. This makes the proposed approach more attractive. First, B$^+$-tree is readily available in all existing commercial database systems. Second, we are not advocating a transformation function that is specially tailored to skyline queries. Instead, as shown in [84, 107],

similar transformation function can be used to support range and nearest neighbor queries. This means that we only need one index structure to support all three types of queries! Before we present the algorithm for progressive skyline computation, we shall illustrate the idea with an example.

Example 3.3. Consider the example shown in Figure 3.5. The data consist of a set of 3 dimensional points whose domains for all dimensions are in the range of [0,1]. If a point's maximum value among all its three dimensions is its first dimensional value, the transformation mechanism will place this point in the first partition. At the end of the transformation, each data point will be located in only one of the partitions, depending on the dimension having the point's maximum value among all its three dimensions. In the figure, we show the content of the partitions after the transformation (without showing the tree structure). Note that in the figure, each partition is sorted in non-ascending order of the maximum value in that dimension. This can be interpreted as scanning the partition from the last leaf node (and backward) within each partition. We note that we will need an additional pointer to the data point if there are other dimensions that are not indexed (i.e., not used in any skyline operation).

dimension 1	dimension 2	dimension 3
(0.9, 0.8, 0.6)	(0.7, 0.8, 0.5)	(0.1, 0.5, 0.9)
(0.9, 0.5, 0.7)	(0.5, 0.8, 0.6)	(0.8, 0.8, 0.9)
(0.9, 0.2, 0.1)	(0.6, 0.6, 0.6)	(0.7, 0.6, 0.9)
(0.8, 0.7, 0.7)	(0.5, 0.6, 0.6)	(0.2, 0.1, 0.9)
⋮	⋮	⋮
(0.2, 0.2, 0.2)	(0.3, 0.4, 0.2)	(0.3, 0.4, 0.5)
(0.2, 0.1, 0.1)	(0.2, 0.4, 0.1)	(0.2, 0.1, 0.3)
(0.1, 0.1, 0.1)	(0.1, 0.3, 0.2)	(0.2, 0.2, 0.3)

Figure 3.5: An example to illustrate the index-based method.

We make several interesting (and important) observations. First, we note that the interesting (and potentially dominant) points are largely at the top. In fact, we can identify some interesting points by simply looking at points with the largest values in each dimension. In our example, there are 7 points (3 from dimension 1,

and 4 from dimension 3) that have the maximum value of 0.9 in some dimensions. Among them, it is clear that (0.8,0.8,0.9) in dimension 3 is in the skyline. This means that we can provide very fast initial response time to the user!

Second, we can prune away some of the points easily without examining them. This follows from the fact that if the minimum value among all dimensions in a point is larger than the maximum value among all dimensions in another point, then the first point dominates the second. Clearly, the larger the minimum value is, the more points we can prune. In our example, clearly (0.8,0.8,0.9) dominates all points whose maximum value is smaller than 0.8. So, all such points need not be examined. Since the structure is organized in sorted order based on the maximum value, this means that we do not need to examine the points to remove them. This translates to saving in I/O cost, and is in contrast with existing algorithms that require the entire database to be scanned at least once.

Third, in the worst case, we can apply existing techniques by scanning the leaf nodes. Even with this strategy, we can expect a gain over existing schemes, since only the dimensions are involved. Fourth, we can clearly optimize the internal structure by ordering the points with the same maximum value by the minimum values. In other words, we further order the points with the same maximum value using values from the other dimensions so that the most *potential* points are analyzed first in each batch. Fifth, unlike sort-based algorithm which may require large main memory (as dominating points can be far apart), the proposed scheme (as noted in the first point) will not suffer the same problem. As such, we expect the scheme to perform well even with a small amount of memory.

Some of the above observations can be summarized in the following results. Theorem 3.4 says that some points can be pruned. Theorem 3.5 says that skyline points can be obtained from points with the largest value. Theorem 3.6 further shows that if we are to examine points in descending order of the largest values, then we can find skyline points (from these points) without considering those with smaller values. This is important for a progressive approach since users will typ-

ically be interested only in the first few skyline points in order get a big picture of the interesting options. Note that it is possible to have multiple points with the same transformed value, and so, the collection of points should be considered collectively when determining the skyline points.

Theorem 3.4. *Consider two points* $x = (x_1, x_2, \ldots, x_d)$ *and* $y = (y_1, y_2, \ldots, y_d)$. *Let* $x_{max} = \max_{i=1}^{d}(x_i)$, $x_{min} = \min_{i=1}^{d}(x_i)$, *and* $y_{max} = \max_{i=1}^{d}(y_i)$. *Let* x_{min} *occurs at dimension* d_{min}, *and* y_{max} *occurs at dimension* d_{max}. *Then, if* $x_{min} > y_{max}$, x *dominates* y.

Proof. Since $x_i \geq x_{min} > y_{max} \geq y_i$, it follows that $x_i \geq y_i$, for all i. Moreover, $x_{min} = x_{d_{min}} > y_{max}$ and $y_{max} = y_{d_{max}} \geq y_{d_{min}}$, it implies that $x_{d_{min}} > y_{d_{min}}$. In other words, there is at least one dimension of x that is better than the corresponding dimension of y. Thus, x dominates y. □

Theorem 3.5. *Let* D *be a database containing* $|D|$ d*-dimensional points. We define* m *as*

$$m = \max_{i=1}^{|D|}(\max_{j=1}^{d} x_{ij})$$

where x_{ij} *corresponds to the value of the* j*th dimension of the* i*th point. We define* M *as follows*

$$M = \{(x_1, x_2, \ldots, x_d) | (x_1, x_2, \ldots, x_d) \in D \wedge \max_{i=1}^{d} x_i = m\}$$

Let S_D *be the skyline of* D, *and* S_M *be the skyline of* M. *Then,* $S_M \subseteq S_D$.

Proof. Let $x = (x_1, x_2, \ldots, x_d) \in S_M$ be an arbitrary skyline point of M. Now, we have $x_k = m$ for some k. For all points in $D - M$, we have

$$m' = \max_{i=1}^{|D|-|M|}(\max_{j=1}^{d} x_{ij})$$

We note that $m' < m$. As such, none of the points in $D - M$ dominates x. This means that x is also a skyline point in D, i.e., $S_M \subseteq S_D$. □

Theorem 3.6. *Let D be a database containing $|D|$ d-dimensional points. Let there be k distinct values in the dimensions of the points in D. Let m_1 denotes the maximum value, m_2 denotes the second largest value, and so on, and finally, m_k denotes the minimum value. Moreover, let us split the database D into k partitions, $P_1, \ldots, P_k$, such that*

$$P_i = \{(x_1, x_2, \ldots, x_d) | (x_1, x_2, \ldots, x_d) \in D \wedge \max_{j=1}^{d} x_j = m_i\}$$

Let S_D be the skyline of D, and S_i be the skyline of P_i. Let us compute S_D by examining partitions in the order P_1, $P_2 \ldots, P_k$. Then, when we are examining P_j, we can determine whether points in S_j are in S_D without having to look at $P_{j+1}, \ldots, P_k$.

Proof (By Induction). The theorem is true for the first step, i.e., examining P_1. This follows from Theorem 3.5. Suppose the theorem is true for step j. Now, consider step $j + 1$. In this case, we already have the skyline points of partitions $P_1, \ldots, P_j$ (after step j). We are now examining partition P_{j+1}. Let $x = (x_1, x_2, \ldots, x_d) \in S_{j+1}$ be an arbitrary skyline point of P_{j+1}. Now, we have $x_q = m_{j+1}$ for some q. For all points in $P_{j+2} \cup \ldots \cup P_k$, we have

$$m' = \max_{i=1}^{|P_{j+2}|+\ldots+|P_k|} (\max_{j=1}^{d} x_{ij})$$

We note that $m' < m_{j+1}$. As such, none of the points in $P_{j+2} \cup \ldots \cup P_k$ dominates x. This means that we do not need to consider partitions $P_{j+2} \cup \ldots \cup P_k$ when we are determining whether points in S_{j+1} are skyline points of D. □

We are now ready to look at the algorithm. Figure 3.6 shows the algorithmic description of our proposed index-based scheme. For clarity, we have assumed that everything fits in memory. In our implementation, should the number of incomparable points be too large to fit into the memory, we have adopted the nested loop approach. Essentially, subsequent incomparable points are written out to a temporary output file. The points in the output file will then be processed in

a subsequent iteration as in the nested loop approach.

The algorithm is highly abstracted. We shall briefly discuss the routines and variables. f_i is a flag that indicates whether dimension i still needs to be searched. When f_i is set to false, it means that all subsequent points are dominated by some point, and so, partition i need not be searched any further. Routine `maxValue(`t`)` returns the maximum value among all dimensions of t. Similarly, `minValue(`t`)` re-

Algorithm Index

```
1.   for i = 1 to d
2.       f_i ← true
3.       t_i ← traverseTreeMax(root, i)
4.       max_i ← maxValue(t_i)
5.       min_i ← minValue(t_i)

6.   mn ← max_{i=1}^{d} min_i
7.   mx ← max_{i=1}^{d} max_i

8.   for i = 1 to d
9.       if mn > max_i
10.          f_i ← false

11.  j ← 1
12.  S ← ∅
13.  while some partitions' f_i value is true
14.      for i = 1 to d
15.          if max_i == mx
16.              P_j ← t_i
17.              S_j ← ∅
18.              t_i ← getNextLeftElement(t_i)
19.              while (maxValue(t_i) == mx)
20.                  mn ← max(mn, minValue(t_i))
21.                  P_j ← P_j ∪ t_i
22.                  t_i ← getNextLeftElement(t_i)
23.              max_i ← maxValue(t_i)
24.      S_j ←computePartitionSkyline(P_j)
25.      S ← S ∪ computeNewSkyline(S_j, S)
26.      j ← j + 1
27.      mx ← max_{i=1}^{d} max_i
28.      for i = 1 to d
29.          if mn > max_i
30.              f_i ← false
```

Figure 3.6: Index-based skyline computation algorithm.

turns the minimum value among all dimensions of t. `traverseTreeMax(`$root$`,`i`)` is a routine that traverses the B^+-tree to obtain the point with the largest value in dimension i. Routine `getNextLeftElement(`t`)` returns the left element of t (if the element is in the left sibling node, then the sibling node will have to be accessed first). Routine `computePartitionSkyline(`P`)` computes the skyline for a set of points P. Any existing algorithms [9] can be used for the computation. In our implementation, we use the block nested loop algorithm. Routine `computeNewSkyline(`S_j`, `$\mathcal{S}$`)` computes the new skyline points to be obtained from S_j and the current skyline points $\mathcal{S}$. Note that the routine also returns these points to the user. It may also involve accessing the data points if not all dimensions are stored at the leaf nodes of the tree.

Steps 1-5 begin the search at the last element of each partition. In step 6, we identify the threshold whereby points are guaranteed to be dominated. This is given by mn, the maximum value among all the minimum values in the dimensions of all seen points. Step 7 provides the value (stored in mx) to identify the group of points whose maximum value among all dimensions must take on. Steps 8-10 do the first pruning to eliminate any partitions that need not be searched. Steps 11-30 proceed on to locate any skyline points as follows. While there are more partitions to be searched (i.e. some partitions' f_i value is true), the search continues by picking the points that have the current maximum value equal to mx and storing them in a separate partition P_j (steps 14-23). At the same time, mn is updated to reflect the maximum value among all the minimum values of the points examined so far. A higher value will result in fewer partitions that need to be searched subsequently. Next, the skyline of the points in the new partition, P_j is determined (step 24). This new set of skyline points are then compared with the skyline points found so far because some of these new skyline points may be dominated by the current list of skyline points (step 25). Finally, the threshold is updated and more dimensions may be eliminated as a result (steps 26-30). The process repeats itself by looking for the next group of points to examine.

3.2.3 Discussion

Both the Bitmap and Index schemes are expected to produce fast initial response time. While Bitmap produces the skyline points one at a time, the Index scheme generates them in bursts (since it examines collection of points together). In terms of overall efficiency, we can expect the Bitmap scheme to be effective for small number of distinct values per dimension, and its performance to degrade as the number of distinct values per dimension increases. Nevertheless, because it is able to return first few answers quickly, it is a competitive algorithm that cannot be ignored totally. On the other hand, the effectiveness of the Index scheme, like all index-based schemes, depends on the selectivity of the skyline operation – if there are a few skyline points, its performance is expected to be superior; otherwise, it may not perform as well. We further note that the scheme requires a B^+-tree index structure to be constructed for every combination of the dimensions of interest. This may lead to the construction of too many indexes which is not feasible. There are ways to extend the scheme so that a single 5 or 10 dimensional index is sufficient but this is beyond the scope of this dissertation.

In our presentation of the proposed schemes, we have imposed some restrictions on the domains of the dimensions, annotations, etc. Here, we shall discuss how the schemes can be easily generalized.

Other Domains

In our discussion, we have assumed that all dimensions have the same domain. Very often, different dimensions will have different domains. In our tourist example, the domain of the `price` dimension is the set of real numbers, while the domain of the `rating` dimension is just the enumerative set containing integers 1 to 5. The Bitmap scheme can handle dimensions with different domains except for those that are non-discrete or have large cardinalities where they can only be supported to a certain extent. For the Index scheme, to handle dimensions with different domains, we need to normalize all domains to a common one, e.g., [0,1]. We note that since

normalization of a dimension relies on the maximum and minimum values of that dimension, adding a new data point with a higher maximum or lower minimum value of that dimension will require that normalized dimension to be renormalized. In view of this problem, we propose that predetermined maximum and minimum values based on domain knowledge be used initially for normalization. These values should be fairly stable i.e. unlikely to change, hence reducing the frequency of renormalization. However, should either the maximum or minimum value changes, then renormalization will be inevitable.

Number of Attributes in a Skyline Query

So far, we have assumed that all the d dimensions of a point are used in the skyline query. Given a point with d dimensions, it is often the case that only d' ($\leq d$) dimensions will be used in the `SKYLINE OF` clause. In fact, we note that the actual number of dimensions used in an arbitrary skyline query may be different at different time and by different user. Let the actual number of dimensions used be d''. Clearly, $d'' \leq d' \leq d$. For the Bitmap scheme, we propose that the bitmap structure be built for all the d' dimensions. For a query that specifies d'' dimensions, the Bitmap scheme requires minimal changes – all we need is to examine only the bit-slices that correspond to the d'' dimensions.

Similarly, for the Index scheme, only values obtained from the d' dimensions need to be transformed and indexed. For a query with d'' dimensions, it depends on whether the index for the d'' dimensions exists. If it does, the Index scheme can be applied directly. Otherwise, alternative algorithms have to be used.

Other Annotations

We have considered only the `MAX` annotation. Here, we discuss how the `MIN` and `DIFF` annotations can be supported.

Let us first examine how to support the `MIN` annotation for the Bitmap scheme. Since the 1s in $BitSlice(x_i, d_i)$ represent points having values $\geq x_i$ for dimension i,

executing a bitwise *not* on it will result in the bit-slice whose 1s represent points having values $< x_i$. On the other hand, since the 1s in $PreSlice(x_i, d_i)$ represent points having values $> x_i$ for dimension i, executing a bitwise *not* on it will result in the bit-slice whose 1s represent points having values $\leq x_i$. Hence, the three key bit-slices S_A, S_B and S_C for a candidate point $x = (x_1, x_2, \ldots, x_d)$ can be computed as follows:

$$\begin{aligned} S_A &= \neg PreSlice(x_1, d_1)\ \&\ \neg PreSlice(x_2, d_2)\ \&\ \ldots\ \&\ \neg PreSlice(x_d, d_d) \\ S_B &= \neg BitSlice(x_1, d_1)\ |\ \neg BitSlice(x_2, d_2)\ |\ \ldots\ |\ \neg BitSlice(x_d, d_d) \\ S_C &= S_A\ \&\ S_B \end{aligned}$$

where $\neg$ represents the bitwise *not* operation. Similar to the `MAX` annotation, if S_C is zero, then x is in the skyline.

For the `DIFF` annotation, it is only meaningful in a skyline query if it is used in conjunction with at least a `MAX` or `MIN` annotation. Otherwise, the standard group-by operation is sufficient. For the Bitmap scheme, no change to the bitmap structure is required. During evaluation, if dimension i has a `DIFF` annotation, then $OrigSlice(x_i, d_i)$ will be used. Recall that $OrigSlice(x_i, d_i)$ is derived by executing a bitwise *exclusive or* on $BitSlice(x_i, d_i)$ and $PreSlice(x_i, d_i)$. We shall now describe how the three key bit-slices S_A, S_B and S_C are computed. For simplicity, we assume that the `MAX` annotation occurs for dimensions 1 to k while the `DIFF` annotation occurs for the remaining dimensions (i.e., $k+1$ to d). To determine whether a candidate point $x = (x_1, x_2, \ldots, x_d)$ is in the skyline, we compute:

$$\begin{aligned} S_A = {} & BitSlice(x_1, d_1)\ \&\ BitSlice(x_2, d_2)\ \&\ \ldots\ \&\ BitSlice(x_k, d_k)\ \& \\ & OrigSlice(x_{k+1}, d_{k+1})\ \&\ \ldots\ \&\ OrigSlice(x_d, d_d) \end{aligned}$$

S_A now has the property that the nth bit is set to 1 if and only if the nth point has values greater or equal to x for each corresponding dimension in the `MAX` annotation,

and has the same value as x for each dimension in the `DIFF` annotation.

$$S_B = PreSlice(x_1, d_1) \mid PreSlice(x_2, d_2) \mid \ldots \mid PreSlice(x_k, d_k) \;\&\; OrigSlice(x_{k+1}, d_{k+1}) \;\&\; \ldots \;\&\; OrigSlice(x_d, d_d)$$

S_B now has the property that the nth bit is set to 1 if and only if the nth point has some dimensions (in the `MAX` annotation) whose values are strictly greater than the corresponding values in x, and has the same value as x for each dimension in the `DIFF` annotation.

Therefore, the resultant bit-slice S_A & S_B has the property that the nth bit is set to 1 if and only if the nth point has each `MAX` annotated dimension's value greater or equal to the corresponding value in x **and** at least one of these values is strictly greater than the corresponding value in x, **and** has the same value as x for each dimension which is `DIFF` annotated. Hence, we can conclude that the nth point dominates x. Conversely, if the resultant bit-slice has a value of zero, no points in the database dominates x and thus, x is in the skyline.

To support the `MIN` annotation for the Index scheme, we can transform the original value. For example, if the domain of dimension i is [0,1], and the value of dimension i for a point is x ($0 \leq x \leq 1$), we can represent x by (1 - x). The proposed scheme remains unchanged. For the `DIFF` annotation, notice that the values for each dimension is already grouped by the B^+-tree structure. Hence, the same algorithm can be used except that the pruning optimization cannot be applied on those dimensions annotated with `DIFF`. Finally, we note that it is not likely that a single dimension be used for both `MIN` and `MAX` annotations. Thus, typically, we only need to keep one partition for the Index scheme. In the unlikely case that a dimension may be used for both, we will have to keep two partitions for this dimension.

Selection Predicates

Our schemes do not deal with any selection predicates directly. Consider our tourist example again; it is likely for our tourist to specify additional predicates in the query, e.g., `price < 200` and/or `rating >= 3`. Clearly, these predicates can be considered while computing the skyline. For the Bitmap scheme, we only need to check the bitmap structure if the point satisfies the conditions; otherwise, they can be pruned away immediately. For the Index scheme, there is also no need to examine points whose values in the dimensions corresponding to the predicates are outside of the targeted range. However, we note that in general, interchanging the order of selection predicates and preference clauses is not a trivial issue [26, 51] and further investigations on this are required as future work.

Updates on Proposed Structures

We now discuss how updates to the data points are managed in our proposed schemes. For the Bitmap scheme, recall that our bitmap structure is augmented with additional information to speed up subsequent processing. Thus, any update operation will generally result in modifications of several bit-slices, and in the worst case, all the bit-slices. First, if a new data point is added, we need to update the bit-slices of each dimension based on the way our augmented bit-slices are constructed. If the bit-slice for a particular value does not exist, we would need to create a new bit-slice for that value and update the rest of the bit-slices accordingly. Second, if a data point is deleted, we do not shift the other data points to fill the gap. Instead, the space is replaced by a "tombstone" in the data file. Then, we set the respective bits for the deleted point to zero and update any affected bit-slices accordingly. Third, for modifications, we first find the bit-slice of the original value and update all the respective bits of the affected bit-slices. Next, we find the bit-slice of the new value and update all the necessary bit-slices. Again, if no bit-slice exists for the new value, a new one has to be constructed. For the Index scheme, since we have chosen to use a B^+-tree for the underlying structure, the insert, modify

and delete algorithms are similar to the B^+-tree algorithms. We note that since this disseration's main focus is on efficiency issues pertaining to the evaluation of preference/skyline queries, maintenance issues are only addressed at a preliminary level here while leaving future work to address such issues in depth.

3.3 Performance Study

To evaluate the effectiveness of our proposed skyline algorithms, we conducted an extensive set of experiments to study their performance. This section reports the experimental setup and our findings.

3.3.1 Experimental Setup

All the experiments are carried out on a Pentium III PC with a 866 MHz processor and 128 MB of main memory running the Linux operating system. We implemented the proposed Bitmap scheme (denoted Bitmap) and the Index scheme (denoted Index). As comparisons, we also implemented the three algorithms proposed in [9]: the block nested loop algorithm where the window is organized as a self-organizing list (denoted BNL), the M-way divide and conquer algorithm (denoted DC), and the B-tree-based scheme (denoted BTree). For BTree, we employ the block nested loop algorithm for cases where it is not possible to determine a skyline point solely through the index. All algorithms are implemented in C++.

The databases used in all our experiments are generated in a similar way as described in [9]. Each database contains 100000 tuples, each of size 100 bytes and are stored in flat files. Each tuple has d attributes and one additional "bulk" attribute that is packed with garbage characters to ensure the tuple is 100 bytes long. For ease of presentation, we shall regard each tuple as a d-dimensional point i.e. we ignore the "bulk" attribute in our discussion. Hence, a tuple of dimension 2 is actually one with 3 attributes (2 attributes for experimental purposes and 1 "bulk" attribute) in our implementation.

However, we differ from [9] in that we use integers instead of doubles. We modified the generator used in [9] to generate integers in the range of [1,100] for our experiments i.e. we restricted the number of distinct values per attribute. Although we are using highly discretized domains, our techniques can be extended to handle continuous domains as in [9]. Three types of databases are generated:

- Independent databases

  Attribute values of tuples in an independent database are generated using a uniform distribution.

- Correlated databases

  In a correlated database, tuples whose values are good in one dimension are also good in other dimensions.

- Anti-correlated databases

  In an anti-correlated database, tuples whose values are good in one dimension are bad in one or all of the other dimensions. For example, prices of land are lower when they are further away from the city center.

In our study, we shall first focus on skyline queries that look for tuples that have high values in all d dimensions i.e., the `MAX` annotation. In other words, a tuple x dominates a tuple y if all d dimensions of x is at least as large as the corresponding dimensions of y and at least one dimension of x is strictly larger than the corresponding dimension of y. This is because we expect the `MAX` and `MIN` annotations to be most frequently used. Towards the end of this section, we will show the results of experiments using a mixture of `MAX` and `DIFF` annotations.

Figure 3.7 shows the sizes of the skylines for different types of databases using different number of dimensions for the `MAX` annotation. From the figure, we observe that the number of skyline points increases as the number of dimensions increases. It is interesting to note that these values are similar to [9] despite the fact that we are using only distinct integer values for the dimensions. An exception is for a 2 dimensional correlated database. Upon investigation, we discovered that this is

due to a larger number of equivalent skyline points in the database. In general, we can expect the number of skyline points to increase as the number of dimensions increases. Furthermore, we can also see that the size of the skyline for correlated databases is fairly small while it is fairly large for anti-correlated databases, with independent databases somewhere in between.

Dimension	Correlated	Independent	Anti-Correlated
2	17	9	35
3	3	15	397
4	6	127	2790
5	9	347	10240
6	36	1328	22716
7	61	2831	38117
8	101	7918	51719
9	185	13223	63782
10	215	22367	73200

Figure 3.7: Skyline sizes for the `MAX` annotation.

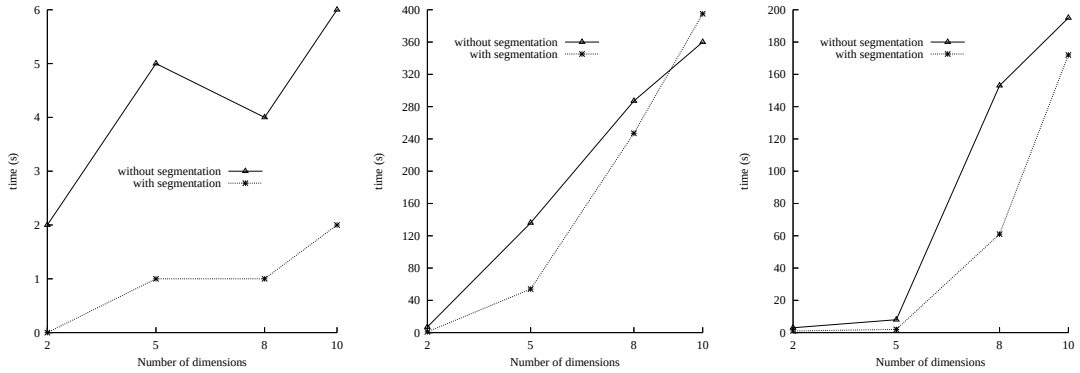
3.3.2 Experimental Results on the `MAX` Annotation

We describe the results of our experiments on the `MAX` annotation in this subsection. Experimental results on other annotations will be discussed in the next subsection.

Experiment 1: Effect of segmentation on the Bitmap scheme

Before we evaluate the performance of the various algorithms, we conducted an experiment to first analyze the effect of segmentation on the Bitmap scheme. For this experiment, we implemented two versions of the Bitmap algorithm – one using segmented bit-slices while the other using unsegmented bit-slices. Furthermore, we included the skyline cache extension in both implementations. We compare the time taken for each type of database (independent, correlated, anti-correlated) using tuples of dimensions 2, 5, 8 and 10 while maintaining 1 MB of main memory throughout the experiments. Figure 3.8 shows the results of the experiment.

From Figure 3.8, we observe that with segmentation, Bitmap performs generally



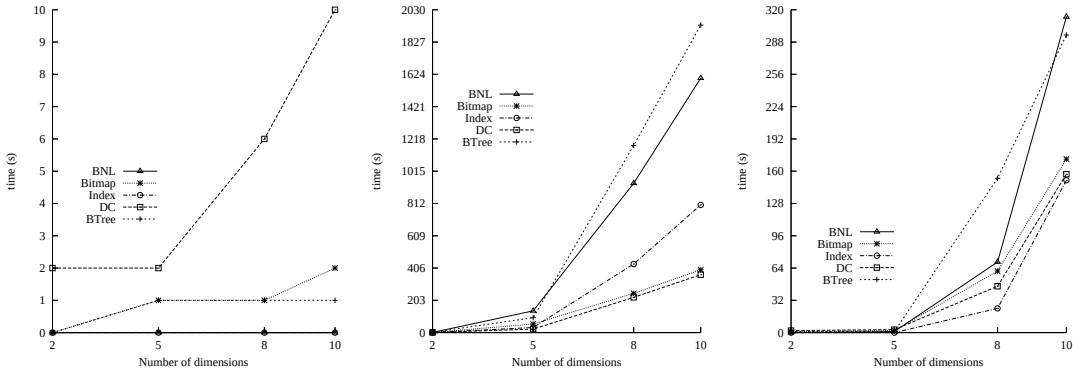
(a) Correlated (b) Anti-correlated (c) Independent

Figure 3.8: Effect of segmentation on the Bitmap scheme.

better. This is clearly illustrated in Figures 3.8(a) and (c) where Bitmap with segmentation outperforms the one without segmentation for all dimensions. However, in Figure 3.8(b), we observe that Bitmap with segmentation performs worse for a 10 dimensional anti-correlated database. This is because in an anti-correlated database, the number of skyline points are significantly higher (especially when the number of dimensions is large). Recall that in the Bitmap scheme with segmentation, if the point under consideration is a skyline point, the Bitmap scheme can only determine this after it has examined all the segments. A larger number of skyline points implies that more segments need to be examined and thus the overheads from accessing the bit-slice index entries as well as the bit-slice segments become significant. However, the overall results do indicate that segmentation has a beneficial effect. Hence, for all subsequent experiments, we use the segmentation optimization as well as the skyline cache extension in our Bitmap scheme.

Experiment 2: Comparing the overall runtime performance

In this experiment, we examine the total amount of time needed by each algorithm (BNL, Bitmap, Index, DC, BTree) to find the skyline. We recorded the time taken by each algorithm for each type of database (independent, correlated, anti-correlated) using tuples of dimensions 2, 5, 8 and 10 while maintaining 1 MB of main memory throughout the experiments. Figure 3.9 shows the results.



(a) Correlated (b) Anti-correlated (c) Independent

Figure 3.9: Actual runtime.

Figure 3.9(a) shows the runtime performance for correlated databases. From the figure, we can see that both BNL and Index perform better than the rest of the algorithms. This is because the number of skyline points in correlated databases is small. This is advantageous for BNL since most of the skyline points can fit into the window and subsequently used for eliminating a large number of tuples from the input. Index is also able to take advantage of this situation. Recall that in our Index scheme, the set of leaf nodes that are processed first usually contain the most dominating points. Thus, when the skyline is small, only a few leaf nodes will need to be scanned, reducing the time taken significantly.

Both DC and Bitmap are not favorable to use in this case because the overheads arising from doing the merging in DC and loading the bit-slices in Bitmap are significant compared to the processing time. Nevertheless, our Bitmap scheme is still substantially better than DC, especially for high dimensions. For BTree, it is only good when the dimensions are small. This is because for correlated databases of low dimensions, the time required to find the first match is relatively small as the values in all the dimensions are fairly close. Its performance deteriorates at higher dimensions due to an increase in the skyline sizes.

A different scenario arises for anti-correlated databases as can be seen in Figure 3.9(b). BNL now performs badly for high dimensions (> 5). Again, this is due to the number of skyline points in the databases. As illustrated earlier, the

number of skyline points in an anti-correlated database is fairly large. This has an adverse effect on BNL, indicating that BNL is only good if the size of the skyline is small. This is consistent with the study done in [9]. We also observe that both DC and Bitmap perform relatively well compared to the rest. On the other hand, Index performs well initially for small dimensions, but decreases as the number of dimensions increases. Recall that Index is highly dependent on the selectivity of the skyline operations. Hence, it is expected to perform badly when there is a large number of skyline points. It is interesting to note that despite the fact that both Bitmap and Index are index-based schemes, their performance do not decrease rapidly when the number of skyline points increases. In particular, Bitmap remains competitively close to DC throughout. As for BTree, although it is still able to perform well for small dimensions, its performance decreases drastically once the number of dimensions exceeds 5. This is inevitable as the attribute values of all dimensions in an anti-correlated database are fairly far apart, thus incurring a high search cost for the first match.

Figure 3.9(c) shows the runtime performance for independent databases of various dimensions. From the figure, we can see that Index now performs the best among all the algorithms. This is a direct consequence of having fewer skyline points in an independent database. The performance of the rest of the algorithms remains relatively unchanged compared to anti-correlated database except that they take shorter time due to fewer skyline points. However, we note an interesting result that differs from the trend at dimension 10, where BTree outperforms BNL. This is because the number of skyline points is nearly double the window size used for BNL. As a result, more than 1 iteration is required to find all the skyline points for BNL. Recall that for BTree, BNL is used as an alternative algorithm when it is not possible to determine the skyline point solely through the index. However, as many tuples are eliminated using the index of the B-tree, the tuples that are subsequently processed by BNL can all fit into the window, resulting in a shorter runtime. We note that such a case does not arise for BTree when the number of

skyline points is very large (as in an anti-correlated database). This is because even after the initial elimination using the B-tree index, the number of tuples that need to be processed subsequently by BNL remains fairly large and usually will require more than 1 iteration, making it performs worse than BNL.

From the results, we can draw the following conclusions. First, in most cases, either Bitmap or Index provides the best performance. The only exception is for the anti-correlated database; however, even in this case, Bitmap is only marginally worse. Second, when the number of dimensions is small or the number of skyline points is small, the Index scheme is superior than the rest of the algorithms. Finally, the Bitmap scheme performs well for large number of skyline points.

Experiment 3: Comparing % of answers returned at intervals

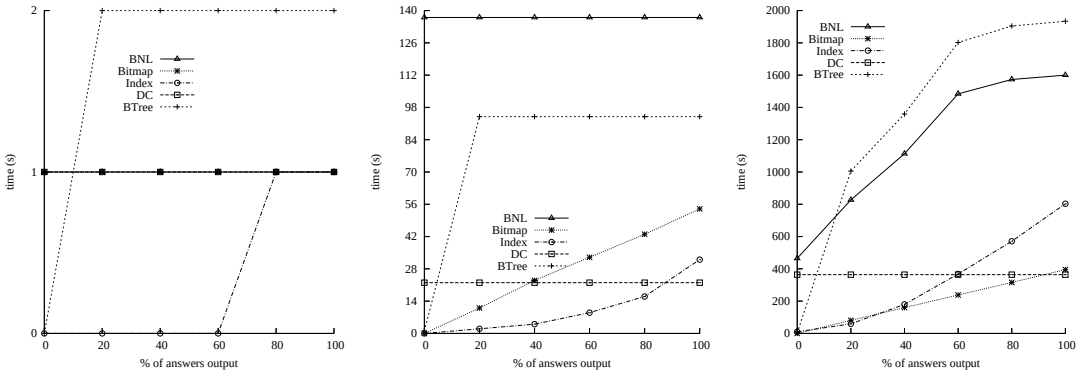
In this experiment, we examine the performance of the algorithms in terms of how fast answers are returned progressively. Recall that the main focus of our two proposed schemes is fast progressive computation of skyline points. This experiment will thus illustrate how effective our algorithms are in terms of producing fast initial response time. Similar to the previous experiment, we tested the algorithms using different types of databases and varying the number of dimensions used while maintaining a buffer size of 1 MB. However, besides keeping track of the overall runtime, we also recorded the time taken for each algorithm to output the first answer (close to 0%) as well as 20%, 40%, 60%, 80% and 100% of the answers. Figures 3.10, 3.11 and 3.12 show our results for anti-correlated, correlated and independent databases respectively.

Figure 3.10 shows the results for anti-correlated databases. From the results, several observations can be made. First, when the number of dimensions is small (< 5), both BNL and Index perform well. This is because the databases with smaller dimensions usually have fewer skyline points, thus enabling BNL and Index to perform faster. Second, when the number of dimensions increases (and hence the number of skyline points), both Bitmap and Index can produce tuples much faster

than the other algorithms. In fact, the first answer from Bitmap and Index is almost instantaneous! This clearly illustrates that both our schemes can progressively compute skyline points much faster than the other algorithms. In particular, our Index scheme is significantly faster than Bitmap when the number of dimensions is small. Bitmap, on the other hand, is useful when the number of dimensions is large although Index is still able to produce the first 40% of the answers marginally faster than Bitmap. Third, DC remains constant for all dimensions. This is because DC is a blocking algorithm and can only start producing answers when it completes its execution. BNL, on the other hand, can start producing answers after the first iteration when all tuples in the database have been examined. However, it is still very much slower than Bitmap and Index which do not require one pass through the database to produce the first few answers. Lastly, although BTree can produce the first tuple much faster than BNL for high dimensions, it is still slow compared to Bitmap and Index. Furthermore, its performance degrades rapidly, making it an undesirable option for progressive skyline computation.

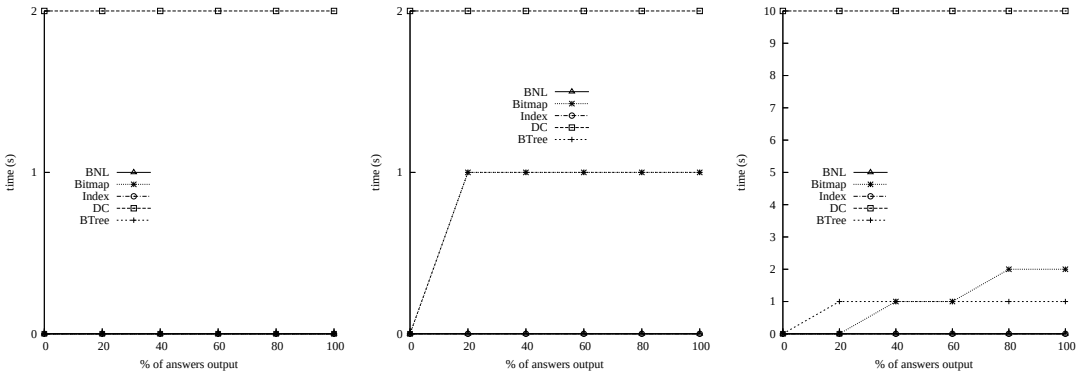
From the results of the experiments on correlated databases (Figure 3.11), we can see that both BNL and Index perform better that DC and Bitmap for all dimensions while BTree is good for dimensions fewer than 5. As mentioned earlier, correlated databases have relatively fewer skyline points. Thus, BNL, Index and BTree perform well in this situation. Bitmap and DC, on the other hand, are much slower due to the overheads involved. However, we observe that Bitmap still performs better than DC for all dimensions. Finally, notice that BTree does not work well for high dimensions (around 10) and may even be outperformed by Bitmap.

Figure 3.12 shows the results for independent databases. First, both Index and Bitmap still produce skyline points progressively faster than the rest of the algorithms. Furthermore, Index now outperforms Bitmap by a wider margin for high dimensions. This is because the number of skyline points is still small for high dimensional independent databases as compared to using anti-correlated databases. Second, both Bitmap and DC perform poorly for low dimensional independent



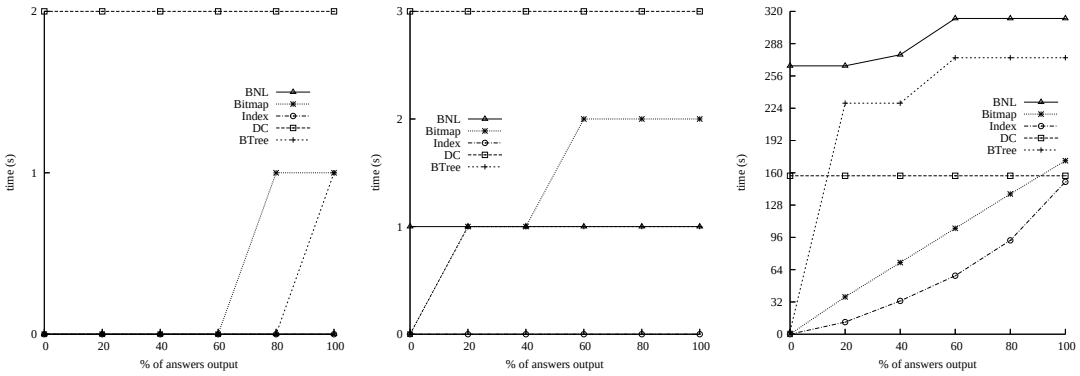
(a) dimension = 2 (b) dimension = 5 (c) dimension = 10

Figure 3.10: Interval timings for anti-correlated databases.



(a) dimension = 2 (b) dimension = 5 (c) dimension = 10

Figure 3.11: Interval timings for correlated databases.



(a) dimension = 2 (b) dimension = 5 (c) dimension = 10

Figure 3.12: Interval timings for independent databases.

databases due to the overheads involved. On the contrary, BNL and BTree perform better for small dimensions but drop drastically as the number of dimensions increases. This is again the direct consequence of larger number of skyline points in high dimensional databases.

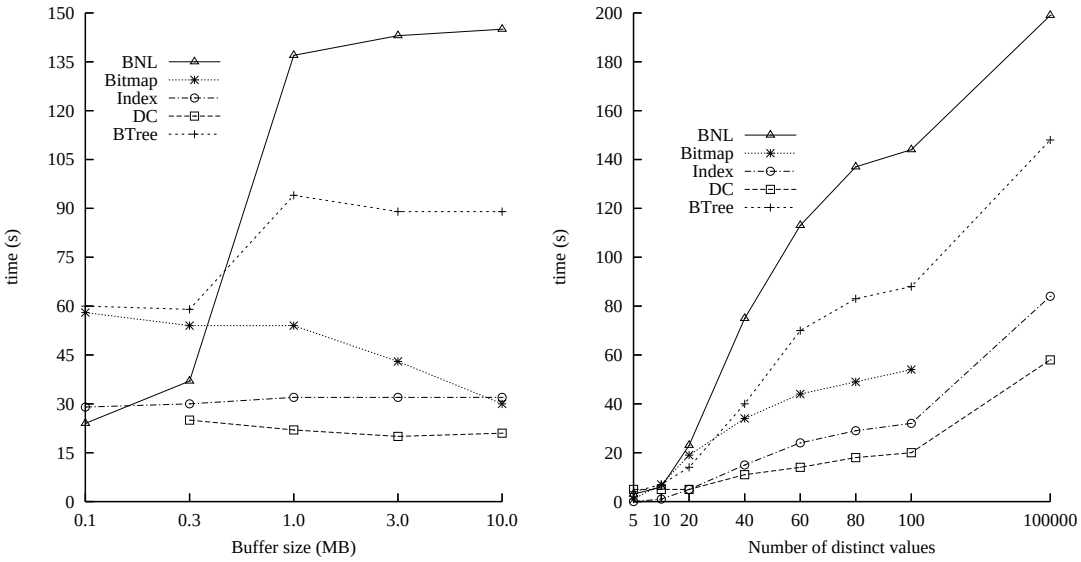
In summary, we believe that Bitmap and Index are useful for progressive skyline computation. In particular, the performance of Index and its robustness to different types of databases of varying dimensions makes it a more attractive option.

Experiment 4: Effect of buffer size

This experiment analyzes the effect of buffer space on the various algorithms. We varied the size of the main-memory buffers from 100 KB (1% the size of the database) to 10 MB (100% the size of the database). For this experiment, we use an anti-correlated database with the dimension of the tuples equal to 5. For DC, because we are using integers as attribute values in our datasets, the partitions created by the algorithm cannot go beyond a certain size and an alternative algorithm, BNL, is used to process such partitions before returning them to DC. This invariably results in very high runtime. Thus, we omit its results for buffer size less than 0.3 MB. Figure 3.13(a) shows the results when the buffer size is varied.

From Figure 3.13(a), we can see that as the buffer size increases from 0.1 MB to 10 MB, the performance of BNL and BTree is good initially, but decreases subsequently. On the other hand, the performance of Index and DC remains fairly consistent while the performance of Bitmap improves as the buffer size is increased. For BNL, a larger buffer implies that the window size is also larger. However, since BNL is CPU-bound, a larger memory actually results in more comparisons, thereby increasing the overall runtime. This is also consistent with the results of [9]. The effect is similar for BTree which makes use of BNL as its alternative algorithm. However, the impact is smaller as a number of tuples are eliminated using the indexes. On the other hand, larger memory is beneficial to Bitmap as more bitslices can be resident in memory. This results in fewer I/Os thereby improving the

overall performance. However, we will not see a big jump in performance since the large size of the bit-slices will cause the extra buffer space to be taken up quickly with just a few bit-slices. For DC, the results actually differ significantly from that of [9] where the performance improves when the buffer size increases. This is because our implementation of DC has to be augmented to deal with integral values for our experiments. Fewer I/Os are incurred by our implementation of DC because we stop the partitioning whenever we hit a partition where all its tuples have the same value in a dimension (which is common when using integral values) and apply a block nested loop algorithm on that partition. Therefore, fewer I/Os are incurred and the effect of memory thus diminishes. Finally, for Index, the increase in memory allows more skyline points to be held in memory, thereby helping to eliminate more tuples without accessing the disk. However, when the memory is large, the skyline points held in memory is also larger and more processing time is incurred. Nevertheless, we note that this effect is minimal to Index, making it a feasible option whether the memory is scarce or not.



(a) Varying size of buffer (b) Varying number of distinct values

Figure 3.13: Effects of buffer size and number of distinct values per dimension.

Experiment 5: Effect of number of distinct values per dimension

In this experiment, we vary the number of distinct values of each dimension. Recall that if the number of distinct values is large, the Bitmap scheme will need to do a lot more processing than the other algorithms. For this experiment, we use a 5 dimensional anti-correlated database and 1 MB of main memory buffer. For completeness, we also include the results where the number of distinct values is 100000 for BNL, Index and DC (Bitmap is not tested for this instance as it is expected to perform badly). This is analogous to the situation where the dimension values are doubles as in [9]. Figure 3.13(b) shows the results.

We can observe that as the number of distinct values increases, the performance of BNL, BTree and Bitmap become worse while the response times of DC and Index just increase slightly. First, when the number of distinct values is small, the number of skyline points decreases. In our experiments, the number of skyline points for 20 distinct values is about 3000 compare to 10000 for 100 distinct values. The smaller number of skyline points enables BNL to perform better than the rest, especially for small number of distinct values. Recall that our Index scheme operates on groups of points (with the same maximum value). Hence, when the number of distinct values is small, the number of points per group increases. It is interesting to note that Index still performs well in this case. This is because even though the initial processing now takes a longer time due to a larger group size, but this also means that the subsequent elimination of a group results in the elimination of more tuples as well. This is why Index still performs well in this case. DC, on the other hand, remains fairly consistent as it is independent of the number of distinct values in the datasets. Finally, Bitmap does not perform as well as we have expected although the runtime has reduced significantly. This is due to the processing overheads. For each tuple, Bitmap has to access the bit-slices for each dimension, which are fairly large and time consuming to process. However, we can expect the performance of Bitmap to improve for even smaller sets of distinct values.

Experiment 6: Effect of database size

In this experiment, we examine the performance of the various algorithms when the size of the database is varied. Figure 3.14 shows the time taken to find the skyline for a data size of 10000 points of a 5 dimensional anti-correlated database. For comparison, we also show the time for the same type of database but containing 100000 points in the same figure. We have also investigated other point sizes but as their relative performance are similar to using 100000 points, we do not show them here.

Algorithm	10000 points	100000 points
BNL	3.0s	137.0s
Bitmap	1.0s	54.0s
Index	2.0s	32.0s
DC	3.0s	22.0s
BTree	2.0s	94.0s

Figure 3.14: Comparing database size (the timings indicate overall runtime).

From Figure 3.14, we observe that there is no significant difference in the relative performance among the algorithms when the database size is small. However, we can see from the results that BNL and BTree do not scale well. On the other hand, DC is able to sustain relatively good performance but it is unable to produce initial answers fast. Our proposed algorithms, especially Index, can handle large databases while maintaining a relatively good performance in terms of progressive computations. Therefore, both the Bitmap and Index schemes remain attractive in environments where datasets are large and anti-correlated and progressive computation is important for the applications.

3.3.3 Experimental Results using `MAX/DIFF` Annotations

The experiments so far use a uniform annotation (`MAX`) for all the dimensions of the skyline queries. We also conducted several experiments to analyze the performance of the various algorithms when the dimensions have different annotations. However,

due to space constraints, we only present an illustrative set of results from these experiments. The first two sets of results compare the overall runtime performance of the various algorithms as well as their ability to provide fast initial response for skyline queries having a different annotation for only one of the dimensions. The last set of results presents our findings for cases where the number of dimensions with a different annotation is more than one.

For simplicity, we only consider two annotations in our skyline queries – `DIFF` and `MAX` (since `MIN` is similar to `MAX`). Figure 3.15 shows the size of the skylines for different types of databases when we restrict the number of dimensions for `DIFF` to 1 while annotating the rest of the dimensions as `MAX`. The last column indicates the number of dimensions used for each annotation. Notice that the number of skyline points has increased significantly compared to the case where `MAX` annotation is used for all dimensions of the queries. This is expected since the number of maximum values in a dimension is fewer than the number of distinct values in the same dimension. Consequently, this leads to more incomparable tuples since two tuples with different values for a dimension with the `DIFF` annotation may both be part of the skyline.

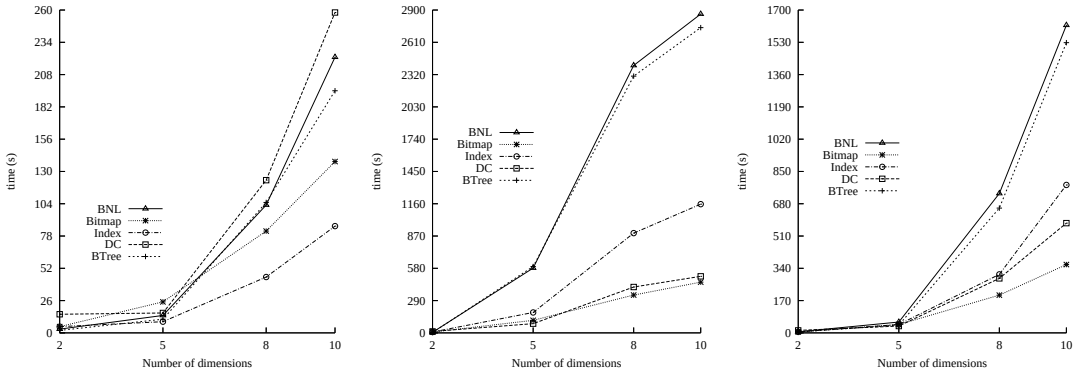
Dimension	Independent	Correlated	Anti-Correlated	Annotations
2	1004	175	470	1 MAX 1 DIFF
5	6033	2524	30339	4 MAX 1 DIFF
8	37403	8828	82394	7 MAX 1 DIFF
10	64093	14435	94258	9 MAX 1 DIFF

Figure 3.15: Skyline sizes (using only 1 `DIFF` annotation).

Experiment 1: Overall runtime performance

Figure 3.16 shows the performance of the various algorithms in terms of overall runtime when we set one of the annotations in the queries to be `DIFF`. We use the same experimental settings as Experiment 2 of the previous subsection. From Figure 3.16(a) which shows the results on correlated databases, we observe that DC, BNL and BTree all perform well initially for small dimensions but deteriorate

rapidly for higher dimensions. In particular, DC's performance is the worst among the three due to the high overheads arising from the partitioning and merging process. Index, on the other hand, outperforms all the other algorithms while Bitmap is worse when the number of dimensions is small, but significantly improved for larger dimensions. This is expected since our earlier experiments have shown that Index performs well when there are fewer skyline points while Bitmap is good when the number of skyline points is high. Lastly, we observe that by setting one of the annotation to `DIFF`, the increase in the number of skyline points has resulted in worse performance for DC, BNL and BTree while both Index and Bitmap can still maintain a relatively good performance.



(a) Correlated (b) Anti-correlated (c) Independent

Figure 3.16: Actual runtime (using 1 `DIFF` annotation).

Figure 3.16(b) shows the results when anti-correlated databases are used. An important point to note is that the number of skyline points is very high, even reaching to 90% of the database size for a 10 dimensional database. First, Bitmap now outperforms the rest of the algorithms when the number of dimensions is high. Recall that in Bitmap, it only needs to find out if any points dominates the one under consideration. As such, it does not require the knowledge of all the skyline points found so far. However, Index, BNL and BTree need to compare with the current set of skyline points found during processing. Hence, if this set is large (as in the case here), it will invariably incur a high runtime penalty. Following the same line of argument, DC which also does not need to compare with the current set of skyline

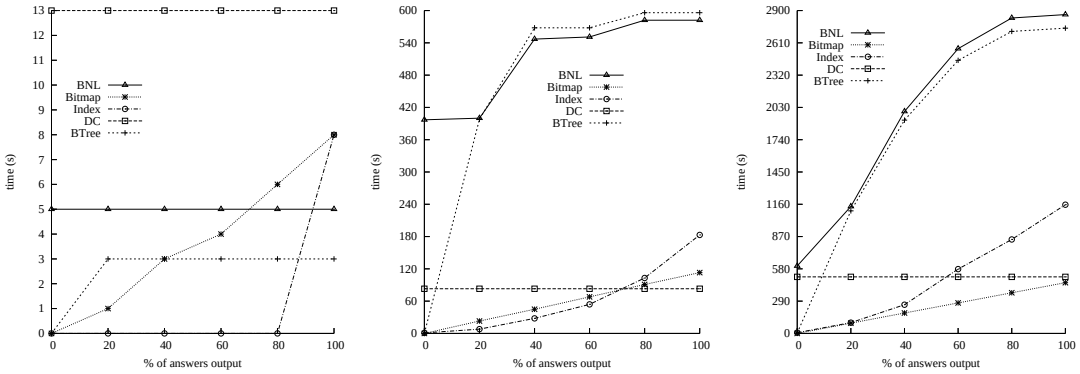
points, is expected to perform well. This is verified by our results. However, since the partition size during merging is now larger, its performance decreases slightly. Finally, we observe from Figure 3.16(c) that the performance remains relatively unchanged for independent databases except that all the algorithms take shorter time and Index's performance is closer to DC due to lower number of skyline points.

In summary, both Index and Bitmap are able to perform well despite using different annotations in the queries. Furthermore, both can outperform the existing algorithms in terms of overall runtime.

Experiment 2: Performance of progressive skyline computation

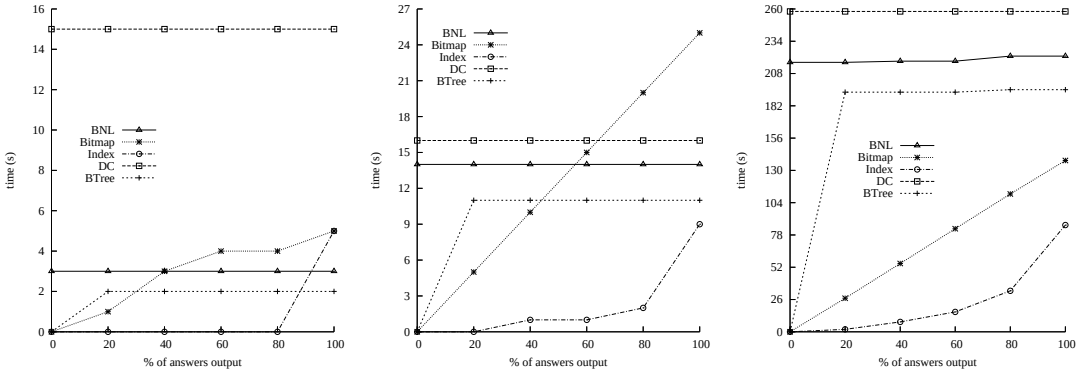
Figures 3.17, 3.18 and 3.19 show the performance of the algorithms in terms of how fast answers are returned progressively. We use the same settings as in the experiments for uniform annotations. We make the following observations. First, both Index and Bitmap are able to progressively output skyline points for all types of databases. More importantly, their ability to provide fast initial answers have not diminished with the inclusion of a different annotation. Second, Index outperforms the rest of the algorithms in terms of returning initial answers when using correlated databases. Its performance deteriorates for high dimensional independent and anti-correlated databases due to a larger number of skyline points in these databases. It is interesting to note that Index is still able to produce the first 20% of the answers quickly under such circumstances. Third, Bitmap, contrary to Index, is able to provide progressively fast computation of skyline points for high dimensional independent and anti-correlated databases. The explanation is similar to that discussed in the previous analysis for overall runtime.

Fourth, because DC is a blocking algorithm, despite its good runtime performance when the number of skyline points is high, it is not feasible in environments that require fast initial answers. Fifth, although BTree is still able to produce the first few answers very quickly, its performance drops drastically, making it an unattractive option. Lastly, BNL is only good when the number of skyline points



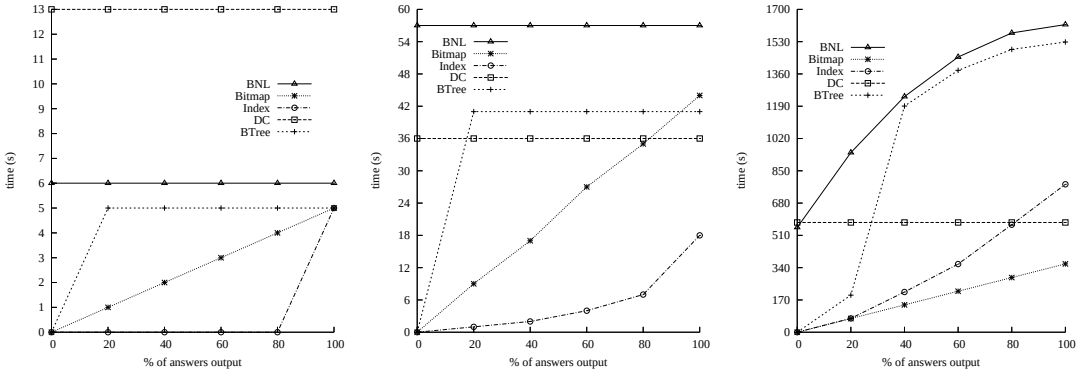
(a) dimension = 2 (b) dimension = 5 (c) dimension = 10

Figure 3.17: Interval timings for anti-correlated databases.



(a) dimension = 2 (b) dimension = 5 (c) dimension = 10

Figure 3.18: Interval timings for correlated databases.



(a) dimension = 2 (b) dimension = 5 (c) dimension = 10

Figure 3.19: Interval timings for independent databases.

is relatively small. Even though it can start producing some answers after the first scan through the database, the initial response still pales in comparison to Index and Bitmap. The above observations clearly illustrate the effectiveness of our algorithms in returning initial answers. More importantly, the experiments show that the use of different annotations has not significantly affected the ability to return answers progressively for both algorithms. Hence, we believe that both Index and Bitmap are attractive solutions for progressive skyline computation, regardless of the annotations used.

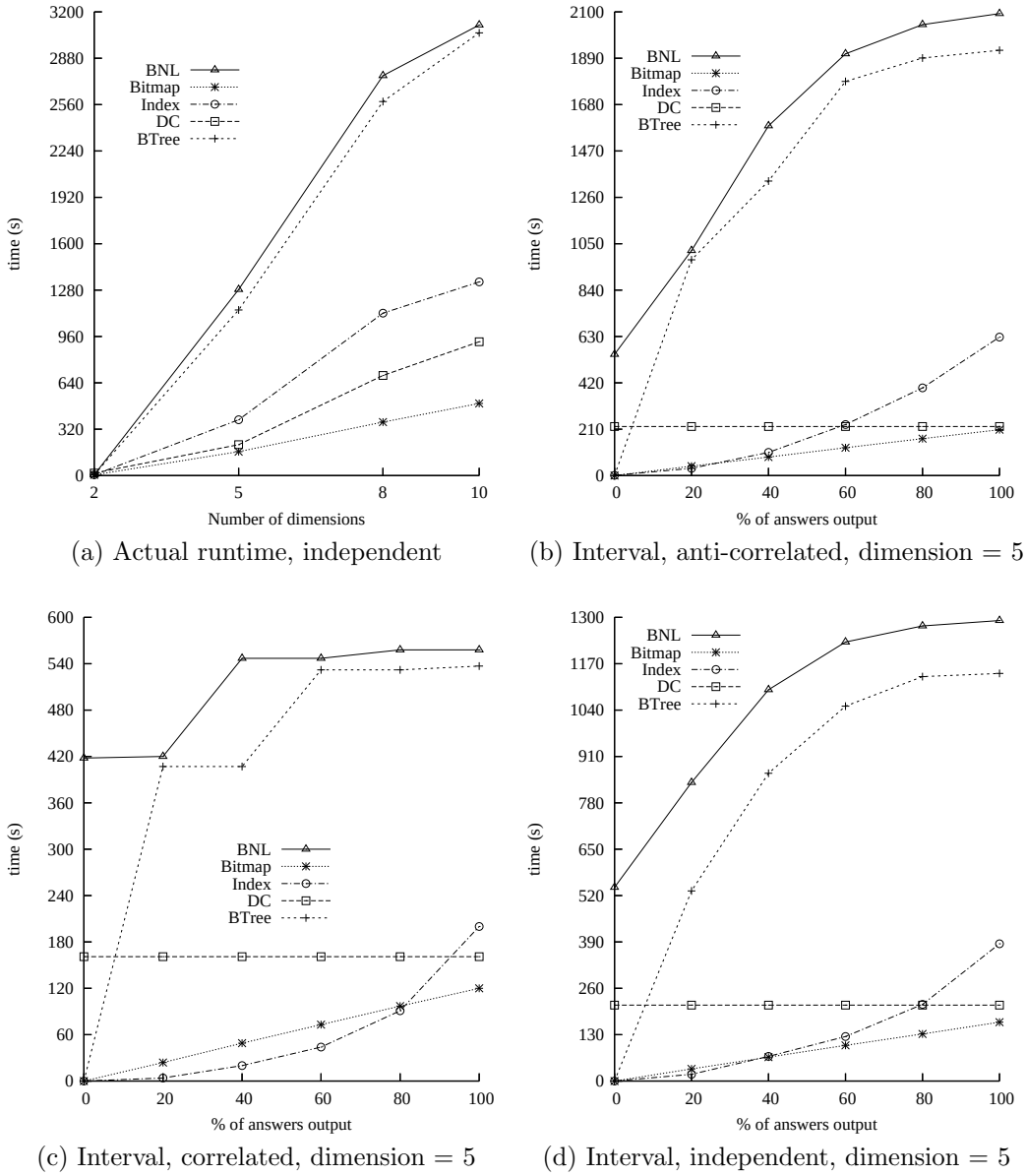
Experiment 3: Using more than 1 `DIFF` annotations

Figure 3.20 shows the sizes of the skylines for various types of databases when more than one dimension of the queries are annotated with `DIFF`. The last column shows the number of dimensions for each annotation. Notice that the number of skyline points has increased tremendously. In particular, nearly all the tuples in a 10 dimensional database are skyline points! Intuitively, we can expect all the algorithms to perform badly. However, we note that it is uncommon for users to pose queries involving more than 1 `DIFF` annotation as it generally results in so many answers that it is hardly useful. Hence, we will only present a representing set of results obtained from the experiments.

Figure 3.21 shows a representative set of results obtained from using queries with more than 1 `DIFF` annotation. Figure 3.21(a) shows the overall runtime performance using independent databases for evaluation. Figures 3.21(b), (c) and (d) show the interval timings using 5 dimensional anti-correlated, correlated and independent databases respectively.

Dimension	Independent	Correlated	Anti-Correlated	Annotations
2	1004	175	470	1 MAX 1 DIFF
5	48815	26116	74455	3 MAX 2 DIFF
8	88342	57093	98252	6 MAX 2 DIFF
10	99924	99068	99989	7 MAX 3 DIFF

Figure 3.20: Skyline sizes (for more than 1 `DIFF` annotations).



(a) Actual runtime, independent

(b) Interval, anti-correlated, dimension = 5

(c) Interval, correlated, dimension = 5

(d) Interval, independent, dimension = 5

Figure 3.21: Using more than 1 `DIFF` annotations.

From Figure 3.21(a), the results remain relatively unchanged from using only 1 `DIFF` annotation in the queries except that the overall runtime has increased significantly. From Figures 3.21(b), (c) and (d), the results do not differ significantly as well. Both Index and Bitmap are still able to provide fast initial response. Furthermore, Index is able to provide the first 20% to 40% of the answers faster than Bitmap. In conclusion, despite using more than 1 `DIFF` annotation in the queries, the overall performance remains relatively unchanged. In particular, Bitmap becomes a more attractive option in the face of higher number of skyline points

resulting from using more `DIFF` annotations in the queries.

3.4 Summary

In this chapter, we have presented two novel algorithms to compute the skyline of a set of points. The main feature of the algorithms is that they can produce skyline points progressively. The first algorithm, the Bitmap scheme, is completely non-blocking and exploits a bitmap structure to quickly identify whether a point is an interesting point or not. The second method, the Index scheme, exploits a transformation mechanism and a B^+-tree index to return skyline points in batches.

Our extensive performance study shows that the proposed algorithms provide quick initial response time as compared to existing algorithms. Moreover, both schemes can also outperform existing techniques in terms of total response time. While the Index scheme is superior in most cases, the Bitmap scheme performs well when the number of distinct values per dimension is small (<10) as well as when the number of skyline points is large.

CHAPTER 4

Skyline Computation with Partially Ordered Domains

In the last chapter, we propose two schemes for progressive skyline computation. These schemes as well as existing techniques for skyline computation all require that the domains of the attributes in the skyline query to have a natural ordering e.g. integers or floats. Partially-ordered attribute domains which include interval data (e.g., temporal data), type/class hierarchies, and set-valued domains, have not been considered. For example, a hotel may also store a set of interesting places within its vicinity, and a tourist may prefer a hotel that is more centralized, i.e., a hotel containing a superset of interesting places or amenities (e.g., gift shop, gymnasium, saloon, sauna, etc.) is preferred.

As another example, categorical data involving roles are typically partially ordered, e.g., in an employee table, there is a hierarchy of reporting structure (project member reports to their project leader who in turn is accountable to the department head and so on) as well as incomparable roles (while the Heads of the manufacturing department and the finance department report to the president of the organization, they do not dominate one another).

For totally-ordered attribute domains, our proposed schemes in the previous chapter as well as recent index-based algorithms like NN algorithm [69] and BBS algorithm [85] have been shown to be superior over the nested-loop approach. How-

ever, because of the lack of a total ordering for partially-ordered attribute domains, it is unclear if index-based schemes can still maintain their competitiveness given that their effectiveness to prune the search space are reduced. To the best of our knowledge, this issue has not been investigated by any of the previous related work.

In this chapter, we address the novel and important problem of evaluating skyline queries involving partially-ordered attribute domains. This chapter is organized as follows. We motivate our approach in the first section. Section 4.2 is the main section of the chapter where we present our evaluation algorithms. We present our experimental results in section 4.3. Finally, we conclude with a summary of our results in the last section.

4.1 Motivation

In this section, we consider the possible evaluation strategies and motivate our proposed algorithms for processing skyline queries with partially-ordered attribute domains. For convenience, we refer to such queries as ***P**artially-**O**rdered **S**kyline queries* (or POS-queries) in contrast to the ***T**otally-**O**rdered **S**kyline queries* (or TOS-queries) that involve only totally-ordered attribute domains.

The most direct method to process POS-queries is to apply the well-known block nested loop approach (BNL) [9], which is the simplest and most versatile approach that works for all types of attribute domains. However, the performance of BNL has been shown to be inferior to that of index-based approaches (as shown in the previous chapter) due to the pruning effectiveness of index-based methods. Another limitation of BNL is that it is a "blocking" algorithm and lacks progressiveness (i.e., answers can only be returned after all the skyline points are computed).

Another strategy to evaluate POS-queries is to try to leverage the effectiveness of previous index-based approaches for TOS-queries by first transforming the partially-ordered attribute domains into totally-ordered domains such that the partial ordering of the original domains are "preserved" in the transformed domains. The most obvious transformation technique is to map each partially-ordered at-

tribute domain into a set of boolean attribute domains, as illustrated by the following simple example.

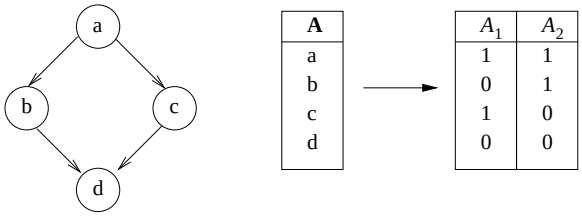


Figure 4.1: Example of domain transformation.

Example 4.1. Consider the simple poset shown in Figure 4.1 for an attribute A consisting of four domain values $\{a, b, c, d\}$ indicated by the nodes in the DAG representation shown. Attribute A can be transformed into a set of two boolean-valued attributes $\{A_1, A_2\}$ depicted by the mapping tables. Each pair of values for A_1 and A_2 are assigned in such a way that two pairs of values do not dominate each other if their original values are at the same level in the DAG and one pair dominates another pair if the first pair's original value is at a higher level in the DAG than the second pair. Thus, given two records r and r', $r.A$ dominates $r'.A$ if r also dominates r' in the transformed domain (i.e., $r.A_1$ dominates $r'.A_1$ and $r.A_2$ dominates $r'.A_2$).

By applying a suitable partial-to-total domain mapping to each partially-ordered attribute, the collection of transformed attributes is now amenable to be indexed using one of the efficient techniques proposed for TOS-queries (e.g., [69, 85]). This transformation is particularly convenient for set-valued attribute domains. However, this boolean transformation suffers from the well-known "dimensionality curse" problem when the size of the partially-ordered attribute domain is large, which will be transformed to a large number of boolean-valued attributes. Thus, the simple boolean mapping is not suitable for index-based methods.

4.2 An Interval-based Approach

To both enable the use of efficient index-based techniques (that are designed for totally-ordered attributes) as well as avoid the "dimensionality curse" problem with using simple domain transformations, the approach that we propose is a "middle-ground" solution that is based on using an approximate, space-efficient domain transformation. In a nutshell, our approach is based on using an approximate *interval* representation (in the form of a pair of integer attributes) for each partially-ordered attribute. This strategy, which increases the dimensionality by one for each partially-ordered attribute, provides a reasonable and practical approximate domain mapping that is amenable to efficient indexing. We shall now present three novel algorithms, namely, BBS^+, SDC, and SDC^+, that are all based on the interval-domain mapping idea to process POS-queries.

4.2.1 Basic Idea

For each partially-ordered attribute A_i with domain D_i, our approach constructs a one-to-one domain mapping f_i that transforms each value $v \in D_i$ into an interval $f_i(v) \in \mathbb{N} \times \mathbb{N}$, where $\mathbb{N}$ denotes the set of natural numbers. The domain mapping f_i is defined such that for any pair of distinct values $v, v' \in D_i$, if $f_i(v)$ contains $f_i(v')$ i.e. the interval $f_i(v')$ falls within the interval $f_i(v)$, then v dominates v'.

Example 4.2. Consider again the poset for attribute A in Figure 4.1. We can construct the following mapping f:

$$f(v) = \begin{cases} [0,3] & \text{if } v = a, \\ [0,2] & \text{if } v = b, \\ [1,3] & \text{if } v = c, \\ [1,2] & \text{if } v = d. \end{cases}$$

Consider values b and c which are dominated by a according to the poset. Using the mapping function f, we can see that their intervals contain the interval of a.

In general, since the transformed values $f_i(v)$ is an approximate representation of the actual values v, it is possible for a pair of transformed attribute values to be incomparable (i.e., neither one of the transformed values contains the other) even though the original attribute values are actually comparable. This implies that when skyline points are computed using the transformed attribute values, it is possible to have *false positives* – points that are considered skylines in the transformed domains but are actually not skylines in the original domains. A false positive is detected when we find that it is dominated by another skyline point based on the original attribute values. While appropriate domain mappings can be constructed for the special case of hierarchical partial orders (i.e., trees) to avoid false positives, false positives are generally inevitable for non-hierarchical partial orders. Therefore, skyline computation algorithms that are based on approximate domain representations need to take into account of false positives. The idea of our proposed algorithms comprises two main steps:

(S1) For each partially-ordered attribute A_i, construct a domain mapping function f_i to transform its domain values v to $f_i(v)$. This effectively replaces each A_i attribute with two integer-domain attributes.

(S2) Organize the transformed data using an efficient indexing method, and use it to compute the skyline points taking into account of possible false positives.

Any existing index-based skyline computation algorithms can be used in our framework. As BBS [85] has been shown to be very efficient for TOS-queries, we adopted it in this work. Our first algorithm, BBS$^+$, which is the least progressive, is a simple extension of BBS that explicitly detects for false positives as the skyline points are computed. Both our second and third algorithms, SDC and SDC$^+$, exploit properties of domain mappings to avoid unnecessary dominance checkings. While SDC stratifies the data at runtime, SDC$^+$ creates the strata offline. SDC$^+$ is the most progressive, and processes the data in stages in such a way that there are no false positives in the intermediate results.

4.2.2 Definitions

We first introduce some notations and definitions. Let $A = \{A_1, A_2, \ldots, A_n\}$ denote the set of attributes of interest, where $A = A_{total} \cup A_{partial}$ with A_{total} and $A_{partial}$ denote, respectively, the subset of totally- and partially-ordered attributes. For each attribute $A_i \in A$, we use $(D_i, \preceq_i)$ to denote the partially order set (or poset) for its domain values D_i. Each $\preceq_i$ is a reflexive, anti-symmetric, and transitive binary relation on D_i. We denote by $\prec_i$ the strict ordering associated with D_i; i.e., $y \prec_i x$ if $y \preceq_i x$ and $x \neq y$. Given $x, y \in D_i$, x and y are said to be *comparable* if either $y \prec_i x$ or $x \prec_i y$; otherwise, they are said to be *incomparable*. We say that x *dominates* y if $y \prec_i x$. A value $v \in D_i$ is a *maximal value* (*minimal value*) if there is no value $v' \in D_i$ such that $v \prec_i v'$ ($v' \prec_i v$).

Consider a finite set of data records R over the set of attributes in A; i.e., $R \subseteq D_1 \times D_2 \times \ldots \times D_n$. Given two records $r_1, r_2 \in R$, we say that r_1 dominates r_2, denoted by $r_2 \prec r_1$, if (1) $r_2.A_i \preceq_i r_1.A_i$ for each attribute $A_i \in A$, and (2) there exists some $A_j \in A$ such that $r_2.A_j \prec_i r_1.A_j$.

Each partial order $(D_i, \preceq_i)$ can be represented by a DAG $G_i = (D_i, E_i)$, where $(v, w) \in E_i$ if $w \preceq_i v$ and there does not exist another value $x \in D_i$ such that $w \preceq_i x \preceq_i v$. For finite domains, this DAG is also known as the Hasse diagram. For simplicity and without loss of generality, we assume that G_i is a single connected component.

For each partially-ordered attribute $A_i \in A_{partial}$ with domain D_i, let $f_i : D_i \rightarrow \mathbb{N} \times \mathbb{N}$ denotes the mapping function constructed for A_i that maps each value $v \in D_i$ into some interval of values (i.e., $f_i(v) = [v_1, v_2]$, $v_1, v_2 \in \mathbb{N}$) such that for any pair of distinct values $v, v' \in D_i$, if the interval $f_i(v)$ contains the interval $f_i(v')$, then v dominates v'.

Based on the transformed values for partially-ordered attributes, we can define a more restrictive form of dominance, called *m-dominance*, as follows. Given two records $r_1, r_2 \in R$, we say that r_1 *m-dominates* r_2, denoted by $r_2 \prec^m r_1$, if (1) $r_2.A_i \preceq_i r_1.A_i$ for each attribute $A_i \in A_{total}$; (2) $f_i(r_2.A_i)$ is equal to or contained

in $f_i(r_1.A_i)$ for each attribute $A_i \in A_{partial}$; and (3) there exists (a) some $A_j \in A_{total}$ such that $r_2.A_j \prec_j r_1.A_j$, or (b) some $A_j \in A_{partial}$ such that $f_j(r_2.A_j)$ is contained in $f_j(r_1.A_j)$. Observe that m-dominance is a stronger form of dominance in that if $r_2 \prec^m r_1$, then $r_2 \prec r_1$; but the converse does not necessarily hold.

The definition of dominance between records can be further generalized to between a record $r \in R$ and a subset of records $e \subseteq R$ as follows: we say that r dominates e (r m-dominates e), denoted by $e \prec r$ ($e \prec^m r$), if r dominates (m-dominates) each record in e.

4.2.3 Domain Mapping Function

For each partially-ordered attribute A_i, the domain mapping function f_i that we use to transform its domain D_i is adapted from the encoding scheme of [1] and works as follows: a spanning tree ST_i is first computed from the DAG G_i, and ST_i is then traversed in postorder with each node v being assigned a unique postorder number $post(v)$. Then, $f_i(v)$ is given by $[x, y]$, where $y = post(v)$ and x is the smallest postorder number assigned to a descendant of v. This mapping scheme satisfies the following *domain mapping property*:

Property 4.1. *If $(v, v') \in E_i$ is also an edge in the spanning tree ST_i, then $f_i(v)$ contains $f_i(v')$.*

It follows that given any two nodes v and v' in G_i, $f_i(v)$ contains $f_i(v')$ iff there is a path from v to v' in the spanning tree ST_i.

Example 4.3. Refer again to the poset for attribute A in Figure 4.1. The domain mapping function f for A is constructed as follows. Let the spanning tree computed from the poset be equivalent to the DAG shown but without the edge (c, d). Then, the postorder numbers assigned to a, b, c, and d are respectively, 4, 2, 3, and 1; and their respective assigned interval values are $[1, 4]$, $[1, 2]$, $[3, 3]$, and $[1, 1]$.

Note that there are other alternative schemes that could be used for the domain mapping function (e.g., [108]). However, as our focus is mainly on skyline compu-

tation algorithms, we have selected a simple mapping function for our work here. Towards the end of this section, we describe how to optimize the spanning tree construction to improve the efficiency and progressiveness of skyline computation.

4.2.4 Algorithm BBS

We shall first review the BBS algorithm which our proposed algorithms are based upon. An overview of the BBS algorithm [85], is shown in Figure 4.2. The set of skyline points is computed by invoking BBS($R, \emptyset$), where R is a R-tree index and S, which represents an intermediate set of computed skyline points, is initialized to the empty set[1]. The algorithm recursively traverses the R-tree, performs

Algorithm BBS(T,S)
Input: T is a R-tree
S is an intermediate set of skyline points
Output: Set of skyline points
1. Initialize heap H to be empty
2. Insert all entries in the root node of T into heap H
3. **while** H is not empty **do**
4. Remove top entry e from H
5. **if** e is an internal entry **then**
6. **if** e is not dominated by any entry in S **then**
7. **foreach** child entry e_i of e **do**
8. **if** e_i is not dominated by any entry in S **then**
9. insert e_i into H
10. **else**
11. S = UpdateSkylines(e,S)
12. **return** S

Algorithm UpdateSkylines(e,S)
Input: e is a data point in some leaf node of a R-tree
S is an intermediate set of skyline points
Output: Return an updated set S
1. **foreach** $p \in S$ **do**
2. **if** e is dominated by p **then**
3. **return** S
4. insert e into S
5. **return** S

Figure 4.2: Algorithm BBS.

[1]We present a slightly more general form of the BBS algorithm (with an input parameter S) to facilitate later presentation of enhanced variants of BBS.

a nearest neighbor search to find regions/points that are not dominated by the current skyline points in S, and inserts these into a main-memory heap structure H. Because BBS visits entries in ascending order of their distances from the origin, each computed point is guaranteed to be a skyline point and can be returned to the user immediately.

In both algorithm BBS as well as our proposed algorithms, we shall refer to the data points maintained in S as *intermediate skyline points*. In the case of BBS (which deals with only totally-ordered attributes), an intermediate skyline point is guaranteed to be a definite skyline point; thus, once a point is inserted into S, it can be output immediately. On the other hand, for two of our proposed algorithms (BBS$^+$ and SDC), the intermediate skyline points in S could be *false positives* and would need to be subsequently detected and eliminated from S. Note that our proposed algorithms are based on the framework of BBS shown in Figure 4.2 with modifications mainly to the `UpdateSkylines` function.

4.2.5 Algorithm BBS$^+$

We shall now present our first algorithm, called BBS$^+$, which represents the simplest extension of BBS to process POS-queries. Algorithm BBS$^+$ is similar to BBS (shown in Figure 4.2) except for the following two changes shown in Figure 4.3. First, since the R-tree index used in BBS$^+$ is based on transformed attribute domains for partially-ordered attributes, the two dominance comparisons in BBS (steps 6 and 8) are replaced with m-dominance comparisons in BBS$^+$. Second, since there could be false positives in the set of intermediate skyline points maintained in S, the `UpdateSkylines` function in BBS$^+$ needs to detect and remove any false positives (steps 4-5) while comparing the new data point e against the intermediate skyline points in S.

Algorithm BBS$^+$(T,S)
Same as Algorithm BBS in Figure 4.2 except that each "dominated" comparison is replaced by a "m-dominated" comparison.

Algorithm UpdateSkylines(e,S)
Input: e is a data point in some leaf node of a R-tree
S is an intermediate set of skyline points
Output: Return an updated set S
1. **foreach** $p \in S$ **do**
2. **if** e is dominated by p **then**
3. **return** S
4. **else if** p is dominated by e **then**
5. delete p from S
6. insert e into S
7. **return** S

Figure 4.3: Algorithm BBS$^+$.

4.2.6 Algorithm SDC

In this section, we present our second algorithm, called SDC, which improves over BBS$^+$ in terms of both speed (by avoiding unnecessary checkings for dominance) as well as progressiveness (by separating the intermediate skyline points into definite skylines and potential false positives).

One major drawback of BBS$^+$ is that it is a non-progressive algorithm due to the possibility of false positives in the computed intermediate skyline points. Another limitation of BBS$^+$ is that it can incur many unnecessary comparisons for dominance; in the worst case, the `UpdateSkylines` function might need to compare the new data point e against every intermediate skyline point in S.

Dominance Classification

To overcome the limitations of BBS$^+$, Algorithm SDC (Stratification by Dominance Classification) exploits two simple characterizations of partially-ordered attribute values based on their domain mapping functions.

Recall that for each partially-ordered attribute A_i, its domain mapping function f_i is defined using the spanning tree ST_i constructed from its partial order DAG $G_i = (D_i, E_i)$. We can classify each value in D_i based on its relationship with

incoming and outgoing values (w.r.t. G_i and ST_i) in two ways as follows. A value $v \in D_i$ is said to be *completely covered* if every directed incoming path to v in G_i is also in ST_i; otherwise, v is said to be *partially covered*. A value $v \in D_i$ is said to be *completely covering* if every directed outgoing path from v in G_i is also in ST_i; otherwise, v is said to be *partially covering*.

Example 4.4. Consider the poset $(D, \preceq)$ with $D = \{a, b, \ldots, j\}$ in Figure 4.4, where the edges included in (excluded from) its spanning tree are indicated by solid (dotted) arrows. The set of values $\{a, b, c, d, f, h\}$ are partially covering; and the set of values $\{f, g, h, i, j\}$ are partially covered.

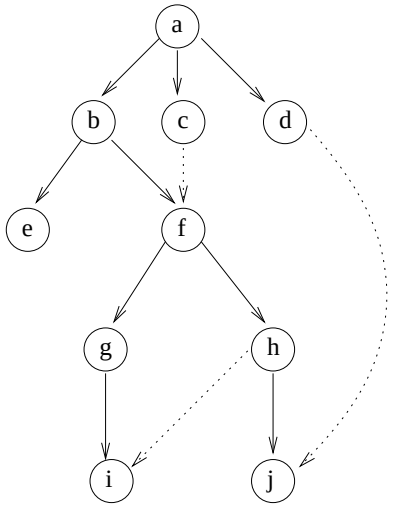


Figure 4.4: Example poset$(D,\preceq)$.

The above classifications of attribute values can be easily generalized to data points as follows. A data point $r \in R$ is said to be *completely covered* if the value of each of its partially-ordered attributes is completely covered; otherwise, r is said to be *partially covered*. Similarly, $r \in R$ is said to be *completely covering* if the value of each of its partially-ordered attributes is completely covering; otherwise, r is said to be *partially covering*.

Based on these two orthogonal classifications, given a set of data points S, S can be partitioned into four disjoint subsets:

$$S = S_{c,c} \cup S_{c,p} \cup S_{p,c} \cup S_{p,p}$$

where each $S_{i,j}$ denotes the subset of points in S that are (1) partially covered (resp. completely covered) if $i = p$ (resp. $i = c$), and (2) partially covering (resp. completely covering) if $j = p$ (resp. $j = c$). The dominance relationship among the four subsets of data points are depicted by the *dominance graph* shown in Figure 4.5.

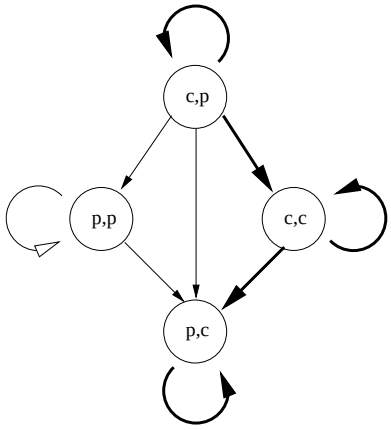


Figure 4.5: Dominance Graph DG.

Lemma 4.1. *A data point $p \in S_{i,j}$ can dominate another data point $p' \in S_{i',j'}$ iff there is a directed edge (normal or bold) from node (i, j) to node (i', j') in the dominance graph DG shown in Figure 4.5.*

Observe that the dominance relationship among the four subsets in Figure 4.5 is reflexive, anti-symmetric, and transitive. The significance of the bold edges will be explained in later subsections. In the following subsections, we present three optimizations used in SDC that are based on the properties of the dominance graph.

Minimizing Dominance Comparisons

To avoid unnecessary dominance comparisons, SDC exploits Lemma 4.1 to organize the intermediate set of skyline points into four subsets. In contrast to BBS$^+$ which compares each new leaf entry e against all the intermediate skyline points in S, SDC only compares e against the necessary subsets of intermediate skyline points.

Referring to SDC's `UpdateSkylines` function in Figure 4.6, step 1 first determines the category, denoted by $S_{i,j}$, of the input leaf entry e. Once this is known,

Algorithm SDC(T,S)
Same as Algorithm BBS$^+$ in Figure 4.3.

Algorithm UpdateSkylines(e,S)
Input: e is a data point in some leaf node of a R-tree
S is an intermediate set of skyline points,
where $S = S_{c,c} \cup S_{c,p} \cup S_{p,c} \cup S_{p,p}$
Output: Return an updated set S
1. Let (i, j) be the category that e belongs, $i, j \in \{c, p\}$
2. Let $C = \{(x, y) \mid$ edge from (x, y) to (i, j) in $DG\}$
3. Let $C' = \{(p, y) \mid$ edge from (i, j) to (p, y) in $DG\}$
4. **foreach** $p \in S_{x,y}$, $(x, y) \in C \cup C'$
5. ret = CompareDominance(e, p)
6. **if** $ret == 1$
7. **return** S
8. **else if** $ret ==$ -1
9. delete p from $S_{x,y}$
10. insert e into $S_{i,j}$
11. **return** S

Algorithm CompareDominance(x,y)
Input: x and y are two data points.
Output: Return -1 if x dominates y, or
1 if x is dominated by y, or
0 if neither x nor y dominates each other.
1. **if** x is m-dominated by y
2. **return** 1
3. **else if** y is m-dominated by x
4. **return** -1
5. **if** x is partially covering **and** y is partially covered
6. **if** x is dominated by y
7. **return** 1
8. **else if** y is dominated by x
9. **return** -1
10. **return** 0

Figure 4.6: Algorithm SDC.

step 2 then selects the categories of data points, denoted by C, that can possibly dominate e (based on Lemma 4.1), and step 3 selects the categories of data points, denoted by C', that e can possibly dominate (to be explained later). Steps 4-9 then compare e against the intermediate skyline points that belong to the selected categories by using an optimized function called `CompareDominance`. This function accepts two input data points x and y and returns -1 if x dominates y, 1 if y dominates x, and 0 otherwise; the details of `CompareDominance` are elaborated in the next subsection.

Optimizing Dominance Comparisons

The second optimization in SDC aims to maximize the use of dominance comparisons that are based on the transformed domains (i.e., m-dominance comparisons) over dominance comparisons that are based on the original domains for partially-ordered attributes. This optimization is useful when dominance comparisons based on the original domains (e.g., set-valued domains) are more expensive to evaluate than dominance comparisons based on the transformed domains which involve two integer comparisons. Therefore, to improve performance for such cases, the more costly dominance comparisons involving the original domains for partially-ordered attributes should be used only as a last resort.

SDC exploits the following property to maximize m-dominance comparisons:

Lemma 4.2. *If x is a completely covering point or y is a completely covered point, then x dominates y iff x m-dominates y.*

This lemma is depicted by the bold edges in Figure 4.5: if $p \in S_{i,j}$, $p' \in S_{i',j'}$, and there is a bold edge from (i, j) to (i', j') in DG, then p dominates p' iff p m-dominates p'.

We briefly explain the correctness of the above lemma. Clearly, if x m-dominates y, then by the domain mapping property, x must necessarily dominate y. On the other hand, if x dominates y, then for each partially-ordered attribute $A_i \in A_{partial}$, there is at least one directed path p from $x.A_i$ to $y.A_i$ in G_i. Since x is a completely

covering point or y is a completely covered point, this implies that the path p must also be in ST_i which means that $f_i(x.A_i)$ contains $f_i(y.A_i)$; therefore, x m-dominates y.

Based on Lemma 4.2, SDC performs dominance comparisons in the function `UpdateSkylines` by using a new function called `CompareDominance` (shown in Figure 4.6). `CompareDominance` first compares x and y using m-dominance, and only when the points are incomparable in terms of m-dominance but could be comparable in terms of dominance (by Lemma 4.2), `CompareDominance` then resorts to comparing them using their original domain values.

Enabling Progressive Computation

The third optimization in SDC aims to enable skyline points to be computed progressively. SDC exploits the following property to identify definite skyline points from the intermediate skyline points.

Lemma 4.3. *An intermediate skyline point that is completely covered is a definite skyline point.*

The correctness of the above lemma is based on the property of the BBS algorithm [85], and Lemmas 4.1 and 4.2. Briefly, an immediate corollary of Lemma 4.1 is that completely covered points (i.e., $S_{c,p} \cup S_{c,c}$) cannot be dominated by partially covered points (i.e., $S_{p,p} \cup S_{p,c}$). Combining this result with Lemma 4.2, it follows from the property of the BBS algorithm [85] that if p and p' are completely covered data points such that p is removed from the heap before p', then p cannot be dominated by p'. This implies that if a newly generated intermediate skyline point is a completely covered point that is not m-dominated by existing intermediate skyline points, then it is necessarily also not dominated by existing intermediate skyline points and it is therefore a definite skyline point.

Therefore, based on Lemma 4.3, the `UpdateSkylines` function in SDC is optimized by checking for false positives only from intermediate skyline points that are partially covered; this explains step 3 of SDC's `UpdateSkylines` which selects

the categories of data points (denoted by C') that the input data point e could dominate. Thus, SDC is more efficient than BBS$^+$ which checks for false positives from all the intermediate skyline points in S.

More importantly, SDC enables the set of skyline points to be computed progressively: each newly determined intermediate skyline point e that is completely covered (i.e., $e \in S_{c,c} \cup S_{c,p}$) can be output immediately since it is a definite skyline point.

4.2.7 Algorithm SDC$^+$

In this section, we present our third algorithm, called SDC$^+$, which aims to further increase the progressiveness of SDC. Recall that SDC essentially organizes the intermediate skyline points into two strata at runtime - the completely covered intermediate skyline points (stratum 1) and the intermediate skyline points that are partially covered (stratum 2). While skyline points in stratum 1 can be progressively returned, those in stratum 2 could be false positives and therefore need to be compared against all the intermediate skyline points to verify that they are indeed definite skyline points. This limitation restricts the progressiveness of SDC since the skyline points in stratum 1 are generally only a small percentage of the entire set of intermediate skyline points found during evaluation (as indicated by our experimental results). To increase progressiveness, SDC$^+$ statically partitions the data into two or more strata.

Data Stratification

In SDC$^+$, the set of data points R is partitioned into a sequence of subsets called *strata* $< R_0, R_1, \cdots, R_k >$ for some value k, such that each $R_i \subseteq R$ and $\bigcup_{i=0}^{k} R_i = R$. By judiciously partitioning the data into separate strata, the skyline points can be computed one stratum at a time starting from R_0 to R_k such that each "local" skyline point in a stratum R_i cannot be dominated by skyline points in the succeeding strata (i.e., R_j, $j > i$), which therefore guarantees that none of the

computed skyline points from each stratum are false positives (as explained earlier). Thus, by computing skyline points from a sequence of smaller subsets instead of from a single large set, the skyline computation becomes more progressive.

An obvious strategy is to organize the data points based on the dominance graph into the following sequence of four strata: $< R_{c,p}, R_{c,c}, R_{p,p}, R_{p,c} >$. However, as the last two strata $R_{p,p}$ and $R_{p,c}$ are generally large which limits progressiveness, SDC$^+$ further refines the last two strata based on the notion of *uncovered level* of attribute values and data points.

We define the *uncovered level* of an attribute value $v \in D_i$, denoted by $L(v)$, as the maximum number of edges in a directed path to v that are in G_i but not in ST_i. The uncovered level of each value v can be computed recursively as follows:

$$L(v) = \begin{cases} 0 & \text{if } v \text{ is a maximal value in } D_i, \\ \max\limits_{(w,v)\in E_i} \{L(w) + \ c(w,v)\} & \text{otherwise.} \end{cases} \tag{4.1}$$

where $c(w,v) = 0$ if (w,v) is an edge in ST_i, and $c(w,v) = 1$ otherwise.

Example 4.5. Consider again the poset $(D, \preceq)$ in Figure 4.4. We have $L(v) = 0$ if $v \in \{a,b,c,d,e\}$, $L(v) = 1$ if $v \in \{f,g,h,j\}$, and $L(v) = 2$ if $v = i$.

The *uncovered level of a data point* r, denoted by $L(r)$, is defined as the maximum of the uncovered levels of its partially-ordered attribute values; i.e.,

$$L(r) = \max_{A_i \in A_{partial}} \{L(r.A_i)\}$$

The notion of uncovered level is useful for refining the dominance relationship among partially covered points as given by the following result.

Lemma 4.4. *A partially covered data point p cannot dominate another partially covered data point p' if $L(p) > L(p')$.*

The correctness of the above lemma follows from the fact that for any pair of attribute values $v, v' \in D_i$, v dominates v' iff there is a directed path from v to v' in G_i. Lemma 4.4 provides a simple and effective way to further partition the data

points in the strata $R_{p,p}$ and $R_{p,c}$ to increase progressiveness. $R_{p,p}$ is partitioned into $k = \max_{r \in R_{p,p}} \{L(r)\}$ strata, where each stratum $R^i_{p,p}$, $1 \leq i \leq k$, represents the subset of data points in $R_{p,p}$ with an uncovered level of i. It follows from Lemma 4.4 that intermediate skyline points from stratum $R^i_{p,p}$ will not be dominated by any data point in $R^j_{p,p}$, $j > i$. Similarly, $R_{p,c}$ is partitioned into $k' = \max_{r \in R_{p,c}} \{L(r)\}$ strata, where each stratum $R^i_{p,c}$, $1 \leq i \leq k'$, represents the subset of data points in $R_{p,c}$ with an uncovered level of i.

Thus, SDC$^+$ partitions the data points in R into $(k + k' + 2)$ strata, where the data points in each stratum $R^i_{x,y}$ is (conceptually) indexed using a separate R-tree $T^i_{x,y}$. The strata is processed in the following sequence: $< R_{c,p}$, $R_{c,c}$, $R^1_{p,p}$, $R^1_{p,c}$, $R^2_{p,p}$, $R^2_{p,c}$, $\ldots >$ as shown in Figure 4.7. Each stratum is processed by calling the function SDC$^+$-sub with two input parameters: the R-tree T for the stratum, and the intermediate set of skyline points S computed so far. The set of skyline points returned by SDC$^+$-sub for the input stratum can be output immediately as they are all definite skyline points. For the special cases of strata $R_{c,p}$ and $R_{c,c}$, the intermediate skyline points (which are completely covered) can be output even earlier: by Lemma 4.3, each intermediate skyline point e is a definite skyline point and can be output before it is inserted into L (step 12 in Figure 4.7). Note that SDC$^+$-sub is similar to BBS$^+$ except for some minor changes.

SDC$^+$ progressively computes the skyline points in ascending order of their uncovered levels starting with the completely covered data points in $R_{c,p}$ and $R_{c,c}$ (which have uncovered level of 0) in steps 1 and 2. Steps 3-7 compute the partially covered skyline points. SDC$^+$ organizes the computed skyline points using two main sets: S stores the definite skyline points computed from the processed strata, while L stores the skyline points computed from the current stratum being processed (which could contain false positives). Thus, the `UpdateSkylines` function in SDC$^+$ processes an input data point e against L and S separately. First, e is compared against L (steps 1-6) to both check if e could be dominated by the points in L as well as check if there are any false positives in L that could be dominated by e.

Algorithm $\mathsf{SDC}^+(T)$
Input: $T = \{T_{c,p}, T_{c,c}, T^1_{p,p} \ldots T^k_{p,p}, T^1_{p,c} \ldots T^{k'}_{p,c}\}$
is a set of $(k + k' + 2)$ R-trees.
Output: Set of skyline points
1. $S = \mathsf{SDC}^+$-sub$(T_{c,p}, \emptyset)$
2. $S = S \cup \mathsf{SDC}^+$-sub$(T_{c,c}, S)$
3. **for** $i = 1$ to $\max\{k, k'\}$
4. **if** $i \leq k$
5. $S = S \cup \mathsf{SDC}^+$-sub$(T^i_{p,p}, S)$
6. **if** $i \leq k'$
7. $S = S \cup \mathsf{SDC}^+$-sub$(T^i_{p,c}, S)$
8. **return** S

Algorithm SDC^+-sub(T,S)
Input: T is a R-tree.
S is an intermediate set of skyline points.
Output: Return an updated set L.
1. Initialize L to be empty
1-10. Same as Algorithm BBS^+ in Figure 4.3 except that in steps 6 and 8, S is replaced by $S \cup L$.
11. L = UpdateSkylines(e,S,L)
12. **return** L

Algorithm UpdateSkylines(e,S,L)
Input: e is a data point in some leaf node of a R-tree.
S is an intermediate set of skyline points,
where $S = S_{c,c} \cup S_{c,p} \cup S_{p,c} \cup S_{p,p}$.
L is the set of skyline points generated from the current stratum.
Output: Return an updated set L.
1. **foreach** $p \in L$
2. ret = CompareDominance(e,p)
3. **if** $ret == 1$
4. **return** L
5. **else if** $ret ==$ -1
6. Delete p from L
7. Let (i,j) be the category that e belongs, $i, j \in \{c, p\}$
8. Let $C = \{(x, y) \mid$ edge from (x, y) to (i, j) in DG, $(x, y) \neq (i, j)\}$
9. **foreach** $p \in S_{x,y}$, $(x, y) \in C$
10. **if** CompareDominance$(e, p) == 1$
11. **return** L
12. Insert e to L
13. **return** L

Algorithm CompareDominance(x, y)
Same as that in Algorithm SDC in Figure 4.6.

Figure 4.7: Algorithm SDC^+.

Next, e is compared against S (steps 7-11) to check if e could be dominated by the points in S. SDC$^+$ uses the same `CompareDominance` function as SDC.

Although each stratum is conceptually indexed independently by a separate R-tree, multiple consecutive strata can actually be indexed using a single R-tree by including an additional stratum number attribute for indexing to facilitate the conceptual approach of processing the sequence of strata.

4.2.8 Optimizing Dominance Classification

In this subsection, we present our final optimization technique, which is applicable to both SDC and SDC$^+$, that aims to reduce the number of dominance comparisons and maximize the use of m-dominance comparisons. The idea is to optimize the construction of the spanning tree for each partially-ordered attribute to maximize the occurrence of certain dominance categories of attribute values over others. The following example illustrates the effect of the spanning tree structure on the dominance classification of the attribute values.

Example 4.6. Consider the two almost similar spanning trees ST and ST' that differ by only one edge for the same DAG in Figures 4.8(a) and (b). The solid edges represent the edges that are in the spanning tree, and the dotted edges represent the edges that are in the DAG but excluded from the spanning tree. Completely and partially covered values are represented by shaded and unshaded nodes, respectively; and completely and partially covering values are represented by nodes with thick and thin lines, respectively. We observe the following differences: (1) b, d, and f are completely covering in ST but partially covering in ST'; and (2) e and g are partially covering in ST but completely covering in ST'.

The above example shows that the dominance classification of the values for an attribute can be varied (to some extent) by changing the structure of the spanning tree constructed from the DAG representation of its poset.

More generally, the structure of the DAG determines whether the nodes are completely or partially covered, while the structure of the spanning tree determines

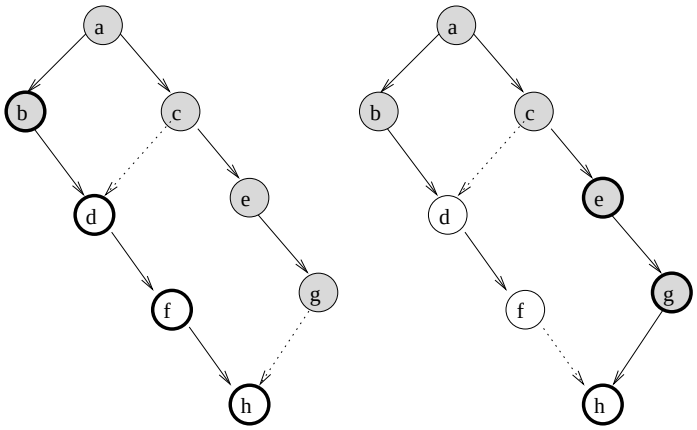


Figure 4.8: Optimizing dominance classification.

whether the nodes are completely or partially covering. Specifically, a node v in a DAG G is a partially covered node in any spanning tree of G if v or an ancestor of v has multiple incoming edges in G; otherwise, v is a completely covered node in any spanning tree of G. The choice of the edges included in the spanning tree will determine whether the nodes are partially or completely covering. In particular, if an edge (v, w) is excluded from the spanning tree, then each ancestor node of v (including v itself) will be partially covering.

Thus, the spanning tree can affect the relative number of nodes between the categories (p, c) and (p, p), and between the categories (c, c) and (c, p). Comparing the two categories (p, p) and (p, c), having more data points in (p, p) relative to (p, c) can reduce the number of dominance comparisons since data points in (p, p) need not be compared against data points in (c, c) (refer to Figure 4.5). On the other hand, having more data points in (p, c) can maximize the use of m-dominance comparisons, whereas comparisons involving data points in (p, p) must be performed in terms of the actual domain values. Thus, the two categories (p, p) and (p, c) have different tradeoffs. Comparing the categories (c, p) and (c, c), having more data points in (c, c) relative to (c, p) is better for performance because it not only reduces the number of dominance comparisons (since points in (c, c) need not be compared against points in (p, p)), but it also enables all the comparisons to be done using m-dominance. Thus, it is better to maximize the number of (c, c) nodes

relative to the number of (p, c) nodes in the spanning tree.

Based on the above analysis, there are two main strategies, referred to as **MinPC** and **MaxPC**, for optimizing the spanning tree construction. In the first strategy, **MinPC**, we minimize the number of (p, c) nodes relative to the number of (p, p) nodes. In the second strategy, **MaxPC**, we maximize the number of (p, c) nodes relative to the number of (p, p) nodes.

For the above strategies, the primary optimization criterion is to minimize (or maximize) the number of (p, c) nodes (relative to the number of (p, p) nodes), and maximizing the number of (c, c) nodes (relative to the number of (c, p) nodes) is used as a secondary criterion (see Figure 4.9). In Fig. 4.8, the spanning trees ST and ST' are created using the **MaxPC** and **MinPC** strategies, respectively.

Note that we have also experimented with two other variations of the above strategies, where the primary and secondary optimization criteria are swapped. Our experimental results indicate that these variations performed worse than their counterpart strategies, showing that minimizing the number of (p, c) nodes is more important than minimizing the number of (c, c) nodes.

Algorithm `OptimizeSpanningTree` in Figure 4.9 takes as input the poset of a partially-ordered attribute, $G = (D, E)$, and computes a spanning tree ST from G that is optimized based on the **MinPC** strategy[2]. Steps 1-6 of the algorithm first initializes the spanning tree ST to be the input DAG G and classifies the nodes into either completely or partially covered nodes, with a default completely covering classification. The classification is computed by a topological traversal of ST since the category of a node depends on the categories of its ancestor (but not descendant) nodes as explained earlier. Next, steps 7-17 then constructs a spanning tree by using a greedy heuristic to delete edges to minimize the number of (p, c) nodes. Here, $parent(v)$ denote the set of parent nodes of a node v in ST. $PCSet_v(w)$ denote the set of nodes in category (p, c) that would become in category (p, p) when all the incoming edges to v, except for (w, v), are deleted from

[2]Changing the comparison operator in step 9 to $\leq$ would result in the **MaxPC** strategy.

ST. In other words, $PCSet_v(w)$ is the set of nodes in category (p, c) such that each node is an ancestor of some node in $parent(v) - \{w\}$. $CCSet_v(w)$ is defined similarly for nodes in category (c, c) that would become nodes in category (c, p).

We use SDC-$MinPC$ and SDC-$MaxPC$, to denote SDC that is optimized using the **MinPC** and **MaxPC** strategies, respectively. SDC$^+$-$MinPC$ and SDC$^+$-$MaxPC$ are defined similarly.

Algorithm OptimizeSpanningTree(G)
Input: $G = (D,E)$ is the DAG representation of a poset
for a partially-ordered attribute.
Output: A spanning tree ST
1. Initialize ST to be G
2. **foreach** node v in ST in topological order
3. **if** v has more than one parent in ST **or**
v's parent is classified as (p, c) in ST
4. Classify v as (p, c)
5. **else**
6. Classify v as (c, c)
7. Let $V = \{v \in D |\ |parent(v)| > 1\}$
8. **while** V is not empty
9. Choose $v \in V$, $w \in parent(v)$ such that
$|PCSet_v(w)| \geq |PCSet_{v'}(w')|$, $\forall v' \in V, w' \in parent(v')$
Break ties by choosing $v \in V$, $w \in parent(v)$ such that
$|CCSet_v(w)| \leq |CCSet_{v'}(w')|$, $\forall v' \in V$, $w' \in parent(v')$
10. **foreach** $u \in parent(v)$, $u \neq w$
11. Delete (u, v) from ST
12. Update u's classification from (x, y) to (x, p)
13. **foreach** $u \in PCSet_v(w)$
14. Update u's classification to (p, p)
15. **foreach** $u \in CCSet_v(w)$
16. Update u's classification to (c, p)
17. Delete v from V
18. **return** ST

Figure 4.9: Algorithm to optimize spanning tree.

4.3 Performance Study

To evaluate the effectiveness of our proposed algorithms, we conducted an extensive set of experiments to study their performance. Our results show that our proposed algorithms (BBS$^+$, SDC, and SDC$^+$) outperform existing techniques by a wide

margin, with SDC$^+$-$MinPC$, which is SDC$^+$ using the **MinPC** strategy to optimize dominance classification, giving the best performance in terms of both response time as well as progressiveness.

Data Sets: We generated synthetic data sets by varying the number of attributes, the correlation among the attributes, the complexity of the posets for partially-ordered attributes, and the size of the data sets. The parameters and their values used are shown in Table 4.1. The first value listed is the default value. In our experimental presentations, default parameter values are used unless stated otherwise.

Parameter	**Values**
$\|A_{total}\|$, # of totally-ordered attributes	2, 1, 4
$\|A_{partial}\|$, # of partially-ordered attributes	1, 2
Attribute correlation	independent, anti-correlated
Poset size (# nodes)	450, 1000
Poset height (# levels)	6, 13
Data size (# data points)	500K, 1000K

Table 4.1: Experimental parameters and values used.

For totally-ordered attributes, we used integer values from the domain $(0, 1000]$, where values are generated as described in [9] with possible correlation among different attributes. For partially-ordered attributes, we used set-valued attributes where dominance is based on set containment. The poset for each partially-ordered attribute is created by first generating a forest of trees, by varying the number of trees, their heights and branching factors. Next, the poset is then formed by randomly connecting nodes among the trees, such that two nodes can be linked only if their levels differ by one. The density of edges in the poset is controlled by the number of iterations of adding inter-tree edges and the probability of adding an edge for a node. The domain of the set-valued attribute values is then derived from the constructed poset. Each data point is generated by choosing a random attribute value from its domain; in particular, for a partially-ordered attribute, a value is selected by randomly choosing a node from its domain's poset.

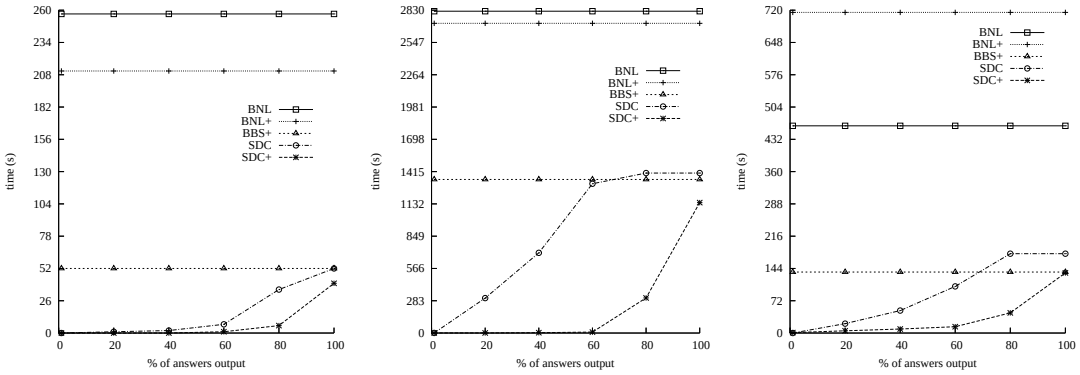
Algorithms: We compared our proposed techniques (denoted by BBS$^+$, SDC, and SDC$^+$), and two variants of the block nested-loop algorithm, denoted by BNL and BNL$^+$. BNL is the basic algorithm proposed in [9], while BNL$^+$ is our optimized extension of BNL that works in a two-stage filter-and-postprocess manner as follows. First, BNL$^+$ executes the standard BNL algorithm (using the transformed attribute values) to quickly obtain a set of intermediate skyline points (possibly with false positives), which is then pipelined to a second BNL algorithm (using the actual attribute values) to eliminate any false positives. We also evaluated the effectiveness of our dominance classification optimization strategies (**MinPC** and **MaxPC**) on SDC and SDC$^+$. For our proposed algorithms, the transformed data values are indexed using R*-trees with page sizes of 4K bytes and node capacity of 50. Each R-tree index is constructed by first scanning the data points to extract the distinct domain values of each partially-ordered attribute. Their posets are then constructed, and each value in each poset is then mapped to an integer interval. Finally, the data points are then indexed with a R-tree on the set of totally-ordered attribute values and the transformed partially-ordered attribute values. Note that each entry in the index nodes has two additional bits indicating whether the entry is partially/completely covered/covering. The posets are therefore not needed once the index is built.

Our experiments were carried out on a Pentium 4 PC with a 2.4 GHz processor and 256 MB of main memory running the Linux operating system.

4.3.1 Response Time & Progressiveness

In this experiment, we examine the performance of the various algorithms in returning first answers as well as how fast *all* answers are returned progressively. For each algorithm, we recorded the time it took to output various percentages of the answers (20%, 40%, 60%, 80% and 100%), as well as the time it took to output the first answer (close to 0%). Although we also captured the time to output every 10 answers, up to 100, we found that the timings are generally the same as the time

to output the first tuple and hence, the time to output the first answer is sufficient to illustrate the response time of the initial set of answers returned. Figure 4.10(a) illustrates the performance when 2 numerical attributes and 1 set-valued attribute are used. In Figures 4.10(b) and (c), we compare the algorithms' performance when the queries consist of more set-valued and numerical attributes respectively.



(a) 2 numerical and 1 set (b) 1 numerical and 2 set (c) 4 numerical and 1 set

Figure 4.10: Varying the number of numerical/set-valued attributes.

From Figure 4.10(a), we can see that SDC and SDC$^+$ have fast initial response times and are progressive. Among them, SDC$^+$ has the best overall performance. There are 662 skyline points and 561 false positives in this experimental run. For BBS$^+$, it is not progressive because it cannot output any answers as they become available due to the possibility of false positives. Instead, each available answer has to be checked against the current skyline using actual set representation to ensure that it is not a false positive. Nonetheless, its performance is still better than the block nested loop algorithms.

For SDC, it is progressive as it can immediately output those intermediate skyline points that are completely covered. Furthermore, to find skyline points efficiently, intermediate skyline points are organized into subsets and m-dominance comparisons are used whenever possible. Comparing with BBS$^+$, this results in a 59% drop in actual set-valued comparisons. Consequently, the initial set of answers can be found very quickly and since most of them are definite skyline points, they can be output immediately. However, as processing continues, its progressiveness

drops as remaining skyline points belong to $S_{p,p}$ and hence, cannot be output immediately. Moreover, since the various subsets are getting bigger as processing continues, this results in more comparisons and hence a poorer performance towards the end.

For SDC$^+$, it is clear that it is more progressive and produces answers faster than SDC. This is because the initial set of strata being processed consists of points belonging only to $S_{c,p}$ and $S_{c,c}$ and hence, any answer found can be output immediately. Since 80% of the skyline points belong to $S_{c,p}$, this explains why SDC$^+$ can output the first 80% of the answers significantly faster. Moreover, we found that SDC$^+$ incurs 30% fewer actual set-valued comparisons and 16% fewer m-dominance comparisons compared to SDC. This is because in SDC$^+$, data points belonging to subsets $S_{c,p}$ and $S_{c,c}$ (which have the highest potential of being in the skyline) are processed first while in SDC, the data points can belong to any subsets. Consequently, more comparisons which do not result in any meaningful outcome are incurred for SDC. Thus, SDC$^+$ has a better overall performance than SDC.

For BNL, its performance is relatively poor throughout as comparing using actual set representation is more expensive than comparing numerical values on the transformed data. This explains why BNL$^+$ can outperform BNL even though it requires a post-processing step.

Consider Figures 4.10(b) and (c) where we increase the number of set-valued and numerical attributes respectively. It is a well known fact in the literature that the number of skyline points increases with increasing number of attributes. For example, with 4 numerical attributes and 1 set-valued attribute, the number of skyline points is 8831 with 9990 false positives. Moreover, adding an additional set attribute increases the skyline points more rapidly than adding a numerical attribute. For example, in Figure 4.10(b), an additional set-valued attribute alone increases the number of skyline points to 9203. As illustrated from these figures, the runtime of the algorithms increases with more numerical/set-valued attributes although their relative performance remain similar. Notice that SDC can be slower

than BBS$^+$ after 60% of the answers are output. Furthermore, its progressiveness drops with increasing number of skyline points. This is because there are now more skyline points belonging to subsets $S_{p,p}$ and $S_{p,c}$ (they cannot be output immediately), resulting in more comparisons and thus, the progressiveness drops. Finally, notice in Figure 4.10(c) that BNL$^+$'s performance is worse than BNL. Since each set-valued attribute is transformed to two numerical attributes, BNL$^+$ now needs to find the skyline for a 6 numerical attributes dataset which by itself, is time-consuming. Coupled with a post-processing step using actual set representation, this results in a poorer performance compared to BNL.

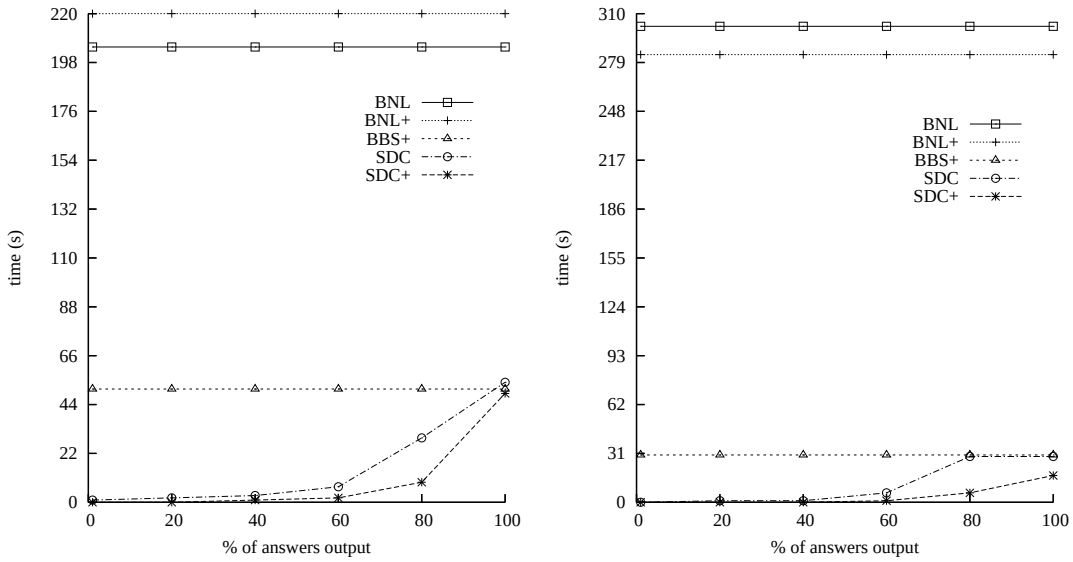
Finally, by looking at the time 100% of the results are produced, we observe the algorithms' performance in terms of overall runtime. We can see that SDC$^+$ has the best overall runtime as it incurs fewer dominance comparisons compared to the rest.

4.3.2 Effect of Poset Structure

Size of poset. Figure 4.11(a) shows the performance of the various algorithms when we increase the size of the poset from the default value of 450 nodes (in Fig. 4.10(a)) to 1000 nodes. We observe that the performance of our proposed algorithms remain relatively unchanged except for a slight increase in runtime for both SDC and SDC$^+$. Increasing the size of the poset has the effect of increasing the number of skyline points. For example, there are 1051 skyline points and 1881 false positives in this experiment. This, in turn, affects the runtime of the algorithms. We can see that BNL$^+$ is most significantly affected by this as it now performs worse than BNL.

Height of poset. Figure 4.11(b) shows the performance of the various algorithms when we increase the height of the poset to 13 by generating a tall and relatively sparse poset to make the number of answers comparable to Figure 4.10(a). This results in 25 strata for SDC$^+$. Again, the relative performance is unchanged compared to previous experiments. However, notice that both BNLand BNL$^+$ has a

higher runtime. This is because a poset with more levels results in sets whose cardinalities are larger. Consequently, the set comparisons become more expensive and this has the largest impact on both BNL and BNL$^+$.



(a) Increasing poset size (b) Increasing poset height

Figure 4.11: Effect of poset structure.

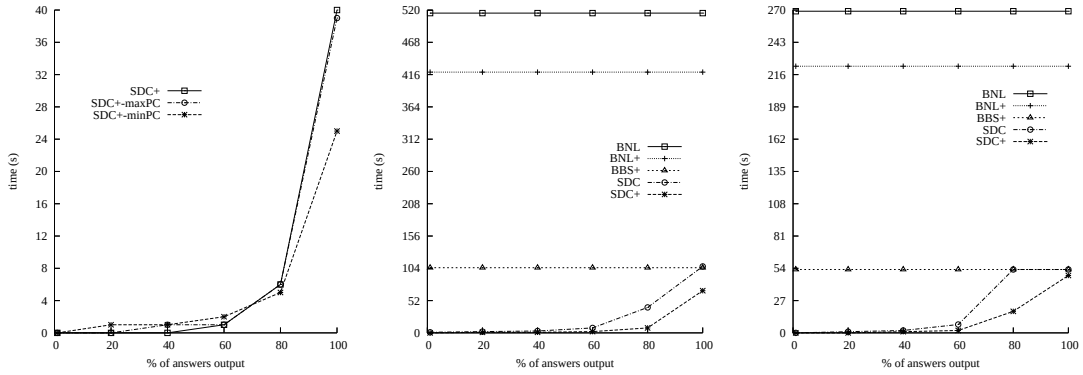
4.3.3 Other Experiments

Effect of Optimized Dominance Classification. Figure 4.12(a) compares the effect of optimizing the dominance classification for SDC$^+$. From the figure, we can see that SDC$^+$-$MaxPC$ has only slight improvement over SDC$^+$ while a more significant improvement is observed in SDC$^+$-$MinPC$. This significant improvement is due to the decrease in the number of dominance comparisons involving data points in the category (c, c). We also conducted experiments comparing SDC against SDC-$MinPC$ and SDC-$MaxPC$, and our results (not shown) indicate that the impact of optimized dominance classification on SDC is not too significant.

Effect of Large Dataset. Figure 4.12(b) shows the results when the size of the dataset is increased to one million data points. We see that the overall runtime for all the algorithms have increased significantly due to the need to process more data tuples. However, both SDC and SDC$^+$ still maintain an advantage over the

rest by being able to produce nearly all the answers before the rest do so.

Effect of Anti-correlated Attributes. Figure 4.12(c) shows the results when the totally-ordered attributes are anti-correlated. This means that if a numerical attribute of a data point has a low value for one attribute, it would have another attribute with high value and so on. From the figure, the relative performance of the various algorithms remains unchanged except for higher runtime. This is because anti-correlation increases the number of skyline points. For example, in this experiment, there are 898 answers compared to 662 when the attributes are independent. With more skyline points, the overall runtime of all algorithms are thus higher compared to the case when independent attributes are used.



(a) Optimizing dominance classification (b) Increasing size of data set (c) Anti-correlated dataset

Figure 4.12: Results of other experiments.

4.4 Summary

In this chapter, we have addressed the novel problem of evaluating skyline queries with partially-ordered attributes. Our solution transforms each partially-ordered attribute into a two-integer domain that enables us to exploit index-based algorithms to compute skyline queries on the transformed space. Based on the framework, we have proposed three novel algorithms: BBS$^+$ is a straightforward extension of BBS, and SDC and SDC$^+$ are optimized versions that handle false positives and facilitates progressive evaluation. Our extensive performance study show that

our proposed algorithms (BBS$^+$, SDC, and SDC$^+$) outperform existing techniques by a wide margin (between a factor of 2 and 16), with SDC$^+$-$MinPC$, which is SDC$^+$ using the **MinPC** strategy to optimize dominance classification, giving the best performance in terms of both response time as well as progressiveness. To the best of our knowledge, this is the first work that examines the problem of evaluating skyline queries with partially-ordered domains.

CHAPTER 5

Evaluating Pareto Preference Queries in Relational Database Systems

In the last two chapters, we examine the evaluation of skyline queries. Although skyline queries is an important class of preference queries, they are limited in expressiveness. Skyline queries only allow users to specify whether they prefer low, high or different values of an entity. For example, a tourist can specify that he prefers *cheap* hotels *close* to the beach but would not be able to express that he also prefers hotels less than 500m away from a metro and preferably affiliated with hotel-chains that he is familiar with (e.g., a Hyatt, Hilton or Sheraton). Such queries, however, can be modelled using the framework described in chapter 2 as *pareto preference queries*.

In this chapter, we examine the efficient evaluation of pareto preference queries in the context of the framework proposed in [65] and described in section 2.1. The pareto preferences we address in our work are based on the set of base preferences proposed in the framework with the exclusion of the EXPLICIT and SCORE base preferences. The EXPLICIT preference is excluded because it is not commonly used in practice while the SCORE preference is more applicable to the quantitative approach which will be addressed in the next chapter.

We propose three approaches for evaluating pareto preference queries efficiently in relational database systems. Our schemes have two notable features:

1. Using the set of base preferences proposed in the framework, our approaches can evaluate any combination of these base preferences expressed as a pareto preference. This means that a broad range of pareto preferences can be supported. Moreover, our approaches also allow preferences to be specified on both ordered and unordered attributes.

2. Unlike existing techniques, our schemes are completely non-blocking and provide a fast initial response time by returning answers as soon as they become available. This allows the users to terminate the processing prematurely as soon as (s)he is satisfied with the partial answers, saving precious resources in computation.

To demonstrate the effectiveness of our approaches, we conducted an extensive performance study comparing our approaches against existing techniques. The results show that our schemes are able to progressively return answers to pareto preference queries efficiently.

For ease of illustration, we shall use the sample hotel relation in chapter 2 as our running example. For convenience, we reproduce the relation in Table 5.1.

Id	Rates	Area	Stars
1	280	midtown	2
2	190	uptown	3
3	308	midtown	3
4	314	midtown	4
5	257	uptown	2

Table 5.1: Hotels relation (from chapter 2).

The attributes consist of the hotel's `id`, the starting `rates` of a room, the `area` the hotel is in and the ratings of the hotel represented by the number of `stars`. The domain of the attribute `area` is {uptown, midtown, downtown} while the domain of other attributes are integers. Throughout our examples, we will consider the evaluation of the following query which in preference SQL [68] is:

```
SELECT     *
FROM       Hotels
PREFERRING rates AROUND(200) AND HIGHEST(stars) AND
           area IN ('uptown')
```

Here, the user is interested in hotels whose rates are *around* $200 but whose ratings are as *high* as possible and preferably *in* the uptown area. As described in subsection 2.1.3 in the example for pareto preferences, hotels 2 and 4 are returned as answers. For ease of presentation of our approaches, we shall assume a database D having d attributes $A = \{A_1, \ldots, A_n, A_{n+1}, \ldots, A_d\}$ and $|D|$ tuples. WLOG, let the first n attributes be *ordered* attributes while the rest are *unordered* attributes.

This chapter is organized as follows. Section 5.1 describes a non-trivial extension of our Bitmap algorithm for evaluating skyline queries to evaluating pareto queries. In section 5.2, we describe a tree-based approach which is more space efficient than the first approach but with a small penalty to the runtime efficiency. Then, section 5.3 describes our third approach which is based purely on single-dimensional indexes and thus provides an easy integration with existing database systems. We present our extensive experimental study in section 5.4 and provide a summary in the last section.

5.1 A Bitmap-based Approach

Our first approach generalizes the bitmap approach described in chapter 3 for evaluating skyline queries. We adopt a similar evaluation strategy by taking each tuple from the relation and determine whether any tuples dominates it from the bitmap structure. A tuple can be returned immediately as an answer if no tuples are found to dominate it, resulting in a fast initial response time.

5.1.1 Construction of the Bitmap Structure

Assume that each attribute A_i has k_i distinct values, $1 \leq i \leq d$. Let a_{ij} denotes the jth distinct value of the ith attribute. WLOG, we assume for each *ordered*

attribute that $a_{i1} > a_{i2} > \ldots > a_{ik_i}$.

The construction of the bitmap structure is similar to that discussed in chapter 3 with one notable difference. Consider a tuple $x = (x_1, \ldots, x_d)$. Again, k_i bits are used to represent x_i. Let the qth bit of the k_i bits correspond to a_{iq} where $a_{iq} = x_i$. First, bits 1 to $q-1$ are set to 0 as usual. Next, we would set bits q to k_i to 1. However, if the attribute under consideration is unordered, then, instead of setting bits q to k_i to 1, we will only set bit q to 1. Furthermore, the resulting bit-slices for unordered attributes are stored arbitrarily as compared to a descending order for ordered attributes.

Example 5.1. Figure 5.1 shows the bitmap structure created for the relation in Table 5.1. Consider hotel 1. Since the rate is 280, the bits corresponding to 314 and 308 are set to 0, while the other bits are set to 1. Similarly, since the value of the `stars` attribute is 2, bits corresponding to values greater than 2 will be set to 0, while the rest are set to 1. Lastly, for unordered attribute `area`, the bit corresponding to value midtown is set to 1 while leaving the rest of the bits 0.

Rates					Area		Stars		
314	308	280	257	190	midtown	uptown	4	3	2
0	0	1	1	1	1	0	0	0	1
0	0	0	0	1	0	1	0	1	1
0	1	1	1	1	1	0	0	1	1
1	1	1	1	1	1	0	1	1	1
0	0	0	1	1	0	1	0	0	1

Figure 5.1: Bitmap example.

Property 3.1, Theorem 3.1, Corollary 3.1, Definitions 3.1 and 3.2 are also applicable here except that they are meaningful only for ordered attributes. However, we augment Definition 3.1 i.e. $BitSlice(a_{ij}, A_i)$ to include unordered attributes. Specifically, if the attribute A_i is unordered, then the 1s in $BitSlice(a_{ij}, A_i)$ represent those tuples having just value a_{ij} for A_i. On the other hand, if A_i is an ordered attribute, then they represent those tuples having values $\geq a_{ij}$. For example, in Figure 5.1, $BitSlice(midtown, area)$ refers to the bit-slice for the attribute

`area` where the 1s represent hotels in the midtown area i.e. `10110`. The following definition is introduced specifically for the evaluation of pareto queries.

Definition 5.1. Let P_i be a base preference specified on attribute A_i. Assume a candidate tuple $x = (x_1, \ldots, x_d)$ where x_i is the value for attribute A_i. We define $BitSlice_{\geq P_i}(x_i, A_i)$ to be the bit-slice whose 1s represent tuples having values a_{ij} for attribute A_i where $x_i <_{P_i} a_{ij} \vee x_i = a_{ij}$ i.e. these tuples have values for A_i that are either better than or equal to x_i with respect to preference P_i. Likewise, we define $BitSlice_{>P_i}(x_i, A_i)$ to be the bit-slice whose 1s represent tuples having values a_{ij} for attribute A_i where $x_i <_{P_i} a_{ij}$ i.e. these tuples have values for A_i that are better than x_i with respect to preference P_i.

5.1.2 Evaluating Pareto Preference Queries

We shall now describe how to use the bitmap structure to evaluate a pareto query. Figure 5.2 is a modified version of the Bitmap algorithm for evaluating a pareto preference query. The query consists of a set of preferences, $P = P_1, \ldots, P_d$, where preference P_i is specified on attribute A_i. & and | represent the bitwise *and* and *or* operations respectively.

The algorithm starts by looping through each tuple in the dataset. The main part of the algorithm (steps 4-7) is to derive bit-slice S_C whose 1s represent tuples in the dataset that dominate the candidate tuple x. The algorithm derives two bit-slices S_A and S_B simultaneously (steps 4-6). S_A is the result of executing a bitwise *and* operation on $BitSlice_{\geq P_i}(x_i, A_i)$ for each preference P_i (step 5) while S_B is the result of executing a bitwise *or* operation on $BitSlice_{>P_i}(x_i, A_i)$ for each preference P_i (step 6). This is followed by executing another bitwise *and* operation between S_A and S_B and assigning it to bit-slice S_C (step 7). Finally, in steps 8-9, the algorithm checks whether S_C is zero. If it is, it tells us that there is no tuple in the dataset that is more preferred than x and we can conclude that x is one of the answers and output it.

The following result demonstrates that given any pareto preference query, the

Algorithm Bitmap
Input: Dataset D, Preferences $P = P_1, \ldots, P_d$
1. **foreach** $x = (x_1, \ldots, x_d) \in D$
2. $S_A \leftarrow 1$ // set each bit of S_A to 1
3. $S_B \leftarrow 0$
4. **foreach** $P_i \in P$
5. $S_A \leftarrow S_A$ & $BitSlice_{\geq P_i}(x_i, A_i)$
6. $S_B \leftarrow S_B \mid BitSlice_{>P_i}(x_i, A_i)$
7. $S_C \leftarrow S_A$ & S_B
8. **if** $S_C == 0$
9. output x

Figure 5.2: Modified Bitmap algorithm.

algorithm correctly derives bit-slice S_C which indicates whether any tuples in the dataset dominates the candidate tuple.

Theorem 5.1. *Consider a dataset D with d attributes $A_1, \ldots, A_d$. Assume a pareto preference query Q specifying preference P_i on attribute A_i for $1 \leq i \leq d$. Assuming that $BitSlice_{>P_i}(x_i, A_i)$ and $BitSlice_{\geq P_i}(x_i, A_i)$ are computed correctly for each preference P_i. Then, the 1s in bit-slice S_C indicate only tuples that dominate the candidate tuple $x = (x_1, \ldots, x_d)$.*

Proof. Let us first examine the properties of bit-slices S_A and S_B:

$$S_A = BitSlice_{\geq P_1}(x_1, A_1) \text{ \& } BitSlice_{\geq P_2}(x_2, A_2) \text{ \& } \ldots \text{ \& } BitSlice_{\geq P_d}(x_d, A_d)$$

where & represents the bitwise *and* operation. Thus, S_A has the property that the nth bit is set to 1 iff the nth tuple, $n = (n_1, \ldots, n_d)$, has the property: $\forall i\ x_i <_{P_i} n_i \vee x_i = n_i$.

$$S_B = BitSlice_{>P_1}(x_1, A_1) \mid BitSlice_{>P_2}(x_2, A_2) \mid \ldots \mid BitSlice_{>P_d}(x_d, A_d)$$

where | represents the bitwise *or* operation. Thus, S_B has the property that the nth bit is set to 1 iff the nth tuple has the property: $\exists k\ x_k <_{P_k} n_k$ for $1 \leq k \leq d$. Next, we examine the property of bit-slice S_C, derived from bit-slices S_A and S_B:

$$S_C = S_A \text{ \& } S_B$$

Thus, S_C, has the property that the nth bit is set to 1 iff the nth tuple has the property: $\forall i\ x_i <_{P_i} n_i \vee x_i = n_i$ **and** $\exists k\ x_k <_{P_k} n_k$ for $1 \leq k \leq d$ i.e. each attribute value of the nth tuple is *better than or equal to* the corresponding attribute value

in x **and** some of its attribute values are *strictly better*. Hence, we can conclude that $x <_P n$. Thus, the 1s in S_C can only represent tuples that dominate x. □

Theorem 5.1 relies on the fact that given a candidate tuple having value x_i for attribute A_i, $BitSlice_{>P_i}(x_i, A_i)$ and $BitSlice_{\geq P_i}(x_i, A_i)$ are derived *correctly* for the preference P_i. We shall now discuss the derivation of these bit-slices for some of the base preferences, focusing on those that will be used later for illustration. The rest are described in Appendix A. Throughout the discussion, we will assume that preference P_i is specified on attribute A_i and x_i is the value for attribute A_i of a candidate tuple.

AROUND. Let P_i be AROUND(A_i,z). If x_i is equal to z, there is no way any tuples can have a value for A_i that is strictly better than x_i. Hence, we set $BitSlice_{>P_i}(x_i, A_i)$ to 0 while $BitSlice_{\geq P_i}(x_i, A_i)$ is equal to $OrigSlice(x_i, A_i)$. If x_i is not equal to z, we derive $BitSlice_{>P_i}(x_i, A_i)$ as follows. First, we compute $distance(x_i, z)$. Then, we retrieve the bit-slice for value a_{iv}, the smallest value $\geq z + distance(x_i, z)$ that exists in the bitmap for A_i. Next, we retrieve the bit-slice for value a_{iu}, the smallest value $> z - distance(x_i, z)$ from the same bitmap. These two bit-slices are illustrated in Figure 5.3 (the black stripes) for the case where the value of a_{iv} is x_i. $BitSlice_{>P_i}(x_i, A_i)$ is then the result of executing a bitwise *exclusive or* operation between these two bit-slices. From Theorem 3.1, the 1s in $BitSlice_{>P_i}(x_i, A_i)$ would represent tuples having values in the range of $[a_{iu}, a_{iv})$ for attribute A_i. In Figure 5.3, values with a corresponding bit-slice in the shaded region fall in the range $[a_{iu}, a_{iv})$. We now show that the distances of these values to z is strictly less than $distance(x_i, z)$.

Theorem 5.2. *Let t be a tuple whose value t_i is in the range of $[a_{iu}, a_{iv})$ for attribute A_i. Then, $distance(t_i, z) < distance(x_i, z)$.*

Proof. First, $a_{iu} \leq t_i < a_{iv}$. Since a_{iu} is the smallest value greater than $z - distance(x_i, z)$ that exists in the bitmap for A_i, $t_i \geq a_{iu} > z - distance(x_i, z)$ i.e. $distance(x_i, z) > -(t_i - z)$. On the other hand, a_{iv} is the smallest value greater

or equal to $z + distance(x_i, z)$ in the same bitmap. Let w be the next smallest value after a_{iv} that exists in the bitmap. If $a_{iv} = z + distance(x_i, z)$, then it is obvious that $t_i < z + distance(x_i, z)$. However, if $a_{iv} > z + distance(x_i, z)$, then $w < z + distance(x_i, z) < a_{iv}$. Since t_i is some value that exists in the bitmap, it must be less than or equal to w. In other words, $t_i < z + distance(x_i, z)$. Hence, we can conclude that if $t_i < a_{iv}$, $t_i < z + distance(x_i, z)$. This means that $distance(x_i, z) > t_i - z$. Therefore, $distance(x_i, z) > distance(t_i, z)$ i.e. the distance of t_i to z will always be shorter than $distance(x_i, z)$. □

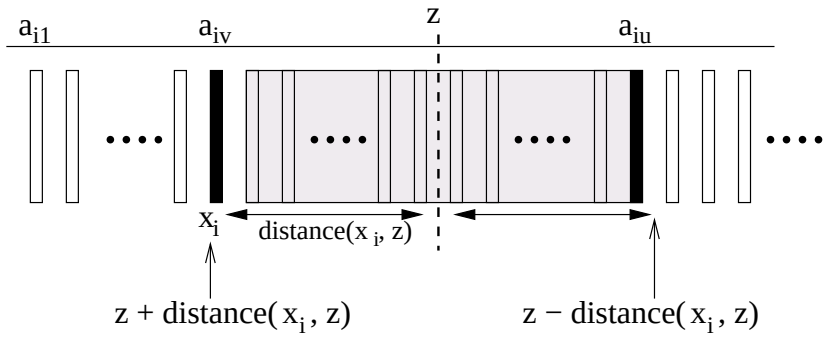


Figure 5.3: AROUND preference.

Finally, to derive $BitSlice_{\geq P_i}(x_i, A_i)$, we simply execute a bitwise *or* operation on $BitSlice_{>P_i}(x_i, A_i)$ and $OrigSlice(x_i, A_i)$. Using the hotel relation in Table 5.1 and the bitmap structure in Figure 5.1, assume that P_i is AROUND(`rate`,200) and the candidate tuple is hotel 1 having rates 280. The distance between the rates of hotel 1 and the desired value is 80. From the bitmap structure, value a_{iu} is given by 190 since it is the smallest value greater than 120 while value a_{iv} is given by 280 since it is the smallest value greater or equal to 280. Hence $BitSlice_{>P_i}(x_i, A_i)$ is given by `10110` $\wedge$ `11111` = `01001` where $\wedge$ is the bitwise *exclusive or* operation. Thus, only hotels 2 and 5 have rates closer to 200 than hotel 1. $BitSlice_{\geq P_i}(x_i, A_i)$ is then given by `01001` | `10000` = `11001`. Now consider hotel 4 whose rates is 314. $BitSlice_{>P_i}(x_i, A_i)$ is given by `00010` $\wedge$ `11111` = `11101`, indicating that hotels 1, 2, 3 and 5 have rates closer to 200 than hotel 4. $BitSlice_{\geq P_i}(x_i, A_i)$ is then given by `11101` | `00010` = `11111`.

HIGHEST. Let the preference P_i be HIGHEST. This is similar to applying the

`MAX` annotation on the attribute A_i as in a skyline query. Hence, as discussed in chapter 3, the 1s in $BitSlice_{>P_i}(x_i, A_i)$ should represent tuples having values $> x_i$ for A_i. Therefore, we can simply retrieve $PreSlice(x_i, A_i)$ since its 1s also represent tuples having values $> x_i$ for A_i. If $PreSlice(x_i, A_i)$ does not exist, then $BitSlice_{>P_i}(x_i, A_i)$ is set to zero. On the other hand, the 1s in $BitSlice_{\geq P_i}(x_i, A_i)$ should represent tuples having values $\geq x_i$ for A_i. This is given by $BitSlice(x_i, A_i)$ as its 1s also represent tuples having values $\geq x_i$ for A_i. As an example, consider the relation in Table 5.1 and the bitmap structure shown in Figure 5.1. Assume that P_i is HIGHEST(`stars`) and the candidate tuple is hotel 1 whose number of stars is 2. $BitSlice_{>P_i}(x_i, A_i)$ is given by $PreSlice(2, stars)$ i.e. `01110` while $BitSlice_{\geq P_i}(x_i, A_i)$ is given by $BitSlice(2, stars)$ i.e. `11111`. Now, consider hotel 4 which has 4 stars. $BitSlice_{\geq P_i}(x_i, A_i)$ is given by $BitSlice(4, stars)$ i.e. `00010` and since there is no preceding bit-slice, $BitSlice_{>P_i}(x_i, A_i)$ is given by `00000`.

POS. Let P_i be POS(A_i,$\{v_1, \ldots, v_m\}$). To derive $BitSlice_{>P_i}(x_i, A_i)$, we first check whether x_i is in the POS-set. If it is, then no tuples can have a value better than x_i for A_i and $BitSlice_{>P_i}(x_i, A_i)$ is set to zero while $BitSlice_{\geq P_i}(x_i, A_i)$ is given by $BitSlice(x_i, A_i)$. However, if x_i is not in the POS-set, the values that are strictly better than x_i are those that are in the POS-set. Hence, $BitSlice_{>P_i}(x_i, A_i) = BitSlice(v_1, A_i) \mid \ldots \mid BitSlice(v_m, A_i)$. In other words, we retrieve the bit-slice of each value in the POS-set and execute a bitwise *or* on them. If a value does not exist in the bitmap, its bit-slice is set to zero. For $BitSlice_{\geq P_i}(x_i, A_i)$, we simply execute a bitwise *or* between $BitSlice_{>P_i}(x_i, A_i)$ and $BitSlice(x_i, A_i)$.

Using the hotel relation in Table 5.1 and the bitmap structure shown in Figure 5.1, let P_i be POS(`area`, {uptown}) and the candidate tuple be hotel 1. Since hotel 1 is located in midtown which is not in the POS-set, $BitSlice_{>P_i}(x_i, A_i)$ is given by the bit-slice for uptown i.e. `01001`, and $BitSlice_{\geq P_i}(x_i, A_i)$ is given by `01001` | `10110` = `11111`. Now, consider hotel 4 located in the midtown area. $BitSlice_{>P_i}(x_i, A_i)$ is also given by `01001` and $BitSlice_{\geq P_i}(x_i, A_i)$ is given by `01001` | `10110` = `11111`. We shall now show the evaluation of a pareto preference query.

Example 5.2. Consider our running example's query: AROUND(`rates`, 200) $\otimes$ HIGHEST(`stars`) $\otimes$ POS(`area`, {uptown}). We shall consider hotels 1 and 4 as candidate tuples. The derivation of $BitSlice_{>P_i}(x_i, A_i)$ and $BitSlice_{\geq P_i}(x_i, A_i)$ for the constituent preferences are shown as examples when describing the derivation of these two bit-slices for the respective base preferences. Consider hotel 1 first. S_A is given by `11001` & `11111` & `11111` = `11001` while S_B is given by `01001` | `01110` | `01001` = `01111`. Hence, S_C is given by `11001` & `01111` = `01001`. Since S_C is not zero, hotel 1 is not an answer. In fact, S_C also indicates that it is dominated by hotels 2 and 5. Next, consider hotel 4. S_A is given by `11111` & `00010` & `11111` = `00010` while S_B is given by `11101` | `00000` | `01001` = `11101`. Then, S_C is given by `00010` & `11101` = `00000`. Hence, we can conclude that hotel 4 is an answer as it is not dominated by any other hotels.

Similar to the original scheme for computing skylines, this approach requires a considerable amount of storage and has high index maintenance costs, especially for dynamic databases. These generally limit the approach to environments such as data warehouses where updates are rare and queries frequent. Nonetheless, its high level of performance (as shown in our experimental study) still makes it the most attractive option in environments where it strives best. In view of the limitations of the bitmap approach, we shall now introduce our tree-based approach, suitable for dynamic databases.

5.2 A R-tree-based Approach

To support preference query processing for dynamic databases, we propose a tree-based approach that is more space efficient than our bitmap approach. The key challenge in designing the tree structure is the need to index both ordered and unordered attributes at the same time. To this end, we propose *Pref-Tree* which stands for **Pref**erence-**Tree**. Although sharing similar features as the R-tree [50], the design of the Pref-Tree is most closely related to the RD-tree [53]. The RD-tree,

a variant of the R-tree, is an index structure for set-valued attributes. The Pref-Tree can also be regarded as a special form of the RD-tree, specifically designed for the efficient evaluation of preference queries.

Preference query evaluation, however, is analogous to the bitmap approach; we take each tuple t of the relation and traverse every branch of the Pref-Tree to the data tuples to check whether any tuple dominates t. If there is none, we can immediately output t, resulting in a progressive approach. Although this approach seems to be computationally expensive, this may not be the case. The search is made efficient using the set inclusion relation that is inherent in the Pref-Tree structure. It allows entire branches of the tree to be rejected during evaluation, thereby effectively reducing the search space. Moreover, through a careful design of the keys in the Pref-Tree, our tree structure is highly *tunable*, allowing a graceful tradeoff between efficiency and index space. We shall begin by discussing our Pref-Tree index structure. This will be followed by a description on how the Pref-Tree index is used to efficiently evaluate pareto preference queries.

5.2.1 The Pref-Tree Structure

Leaf nodes in the Pref-Tree contain entries of the form (*bounding set*, *disk pointer*). The *disk pointer* points to a distinct subset T of the data tuples in D that are stored in secondary storage. The *bounding set* is analogous to the bounding box of a R-tree and consists of d sets $s_1, \ldots, s_d$. To create the set s_i, we take the value of attribute A_i of each tuple $t \in T$ and union them.

Non-leaf nodes contain entries of the form (*bounding set*, *child pointer*). The *child pointer* points to the child node of an entry while the *bounding set* is the union of the bounding sets of all entries in the child node of that entry. Since the attribute values represented in the bounding sets of the leaf nodes are contained in the bounding sets of the non-leaf nodes, the transitive set inclusion relation, that is the key to Pref-Tree's efficiency, exists.

An important issue that needs to be addressed is the representation of the

bounding set. While it may be possible to represent the sets directly by storing the actual attribute values in the nodes, this approach is impractical for large datasets. Furthermore, not all bounding sets are of the same size. This means that keeping the size of the bounding sets fixed may not be space efficient. Based on the criteria described in [53] for key representation, we have to choose a representation that 1) keeps the bounding set small. This allows each node to contain more entries, thereby increasing the fanout and reducing the height of the tree; 2) be as complete as possible i.e. each set should minimize the number of attribute values that do not exist in the data tuples covered by the set; 3) supports computations necessary for evaluating preference queries efficiently.

For the Pref-Tree, we adopt two different representations for the individual sets of a bounding set. One is for representing ordered attribute values while the other is for representing unordered attribute values.

Ordered Attributes. For an ordered attribute, its values are represented as *rangesets* as described in [53]. A rangeset is an ordered list of m disjoint ranges $\{[a_1, b_1],\ [a_2, b_2],\ \ldots,\ [a_m, b_m]\}$ where $a_i \leq b_i$ and $b_i < a_j$ whenever $i < j$. For example, $S = \{[257, 280], [308, 314]\}$. Efficient algorithms for computing rangesets can be found in [53]. Notice that the value of m can be used to tune the Pref-Tree. A large value for m approximates the actual values more closely but increases the size of the node, thereby reducing fanout. Thus, while the algorithm may end up examining more nodes, the search space might be pruned more effectively. On the other hand, a smaller value for m provides a better fanout but results in a less effective pruning of the search space.

Unordered Attributes. For an unordered attribute, its values are represented as *signatures* as described in [61]. Each value is represented by a b-bit signature in which k of the bits are set randomly. A set, S, is then represented by superimposing i.e. taking a bitwise *or* of the signatures of the values in S. We call the resulting signature the *set signature* of S. Given the set signature of S, we can easily determine whether a value x exists in the set S by executing (signature of x &

¬ signature of S) and checking whether the result is equal to zero. A result of zero means that x exists in the set S. Note that & and ¬ represent the bitwise *and* and *not* operations respectively. Although the set signature can be used to represent a lot of values, it has a limitation. We cannot assume that the signatures of distinct sets are also distinct as the combined signature of all values in a set may also include the signature of other values. An example will make this clear.

Consider the sample relation in Table 5.1. Let the value of b and k be 5 and 2 respectively and the signatures of the domain values of attribute `area` are as follows: uptown is `10010`, midtown is `00101` and downtown is `00110`. If the set S is {uptown, midtown}, then its set signature would be `10010 | 00101 = 10111`. Notice that the set signature of S also includes the signature of downtown. These extra values are known as *false drops*. As we will see later, the presence of false drops has a bearing on efficiency as it can lead the search into paths which would be avoided if false drops are not possible.

Similar to using rangesets, b and k can be used to tune the Pref-Tree. Different values of b and k will result in different probabilities of false drops occurring. On one hand, we can choose a high value for b and a relatively low value for k so that chances of false drops are lower. However, this would increase the size of an entry, reducing the fanout of the tree. On the other hand, bad values for b and k can result in many false drops, reducing the effectiveness of the Pref-Tree. In summary, by using different number of ranges in a rangeset or different values b and k for a signature, a graceful tradeoff between efficiency and index size can be achieved.

Example 5.3. Figure 5.4 shows the Pref-Tree structure for the hotel relation in Table 5.1. We have assumed an artificial number of 2 entries per node and 1 range per rangeset. For ease of illustration, we assume that there are no false drops in the representation of the unordered attribute values and show the actual values rather than the signatures. Consider the first entry of node N11 (on the left). The range of the `rates` attribute is computed by merging the ranges [280,280] and [257,257] of the entries in node N111. This is done similarly for the `stars` attribute. However,

since both ranges for the `stars` attribute in node N111 are the same, the resulting range for the `stars` attribute in node N11 is also [2,2]. As for the attribute `area`, the union of values midtown and uptown in node N111 is stored in node N11. In the actual representation, a bitwise *or* operation is executed on the signatures of midtown and uptown and the combined signature stored in node N11.

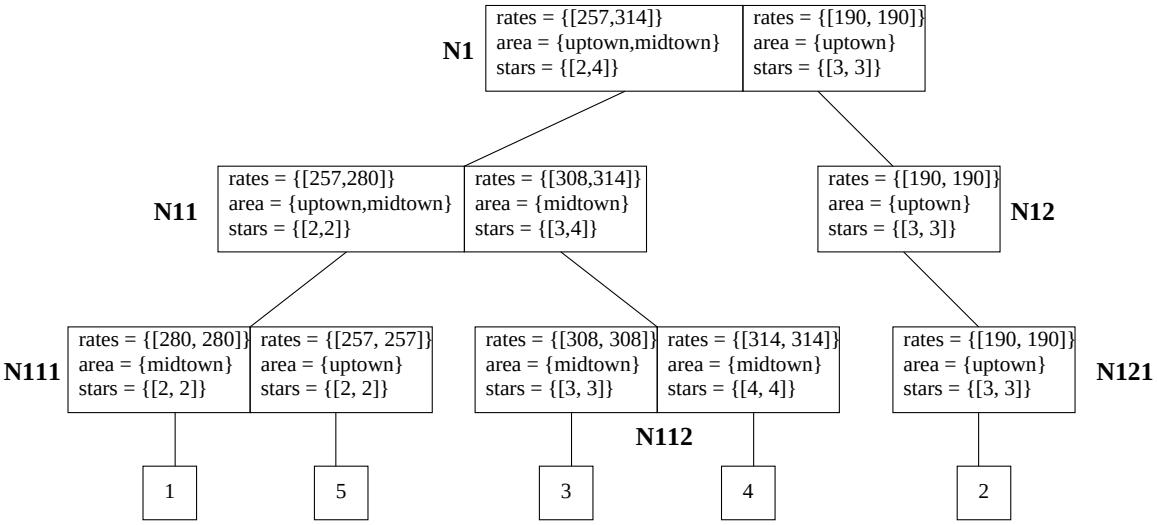


Figure 5.4: Pref-Tree example.

5.2.2 Insertion and Deletion Operations

We shall now briefly describe the insertion and deletion operations of the Pref-Tree. We do not show the algorithms as they are analogous to the R-tree and RD-tree. To insert a new tuple, t, the insertion algorithm starts by examining the bounding set of each entry of the root node. For each attribute A_i, the respective attribute value, t_i of t, is compared against the set s_i of the bounding set. If s_i is a rangeset, the algorithm computes the shortest distance between t_i and each range in the rangeset. For a range $r = [a,b]$, the distance between t_i and r is given by $min(abs(a - t_i), abs(b - t_i))$. In the case where $a \leq t_i \leq b$, the distance is zero. Intuitively, the distance is also the least enlargement required for this rangeset. On the other hand, if s_i is a set signature, the algorithm simply checks whether t_i exists from the signature. If t_i exists, it sets the distance to 0, 1 otherwise. After the algorithm computes the respective distance for each attribute of each entry,

a skyline operation is executed to eliminate those entries that are dominated by some other entries (s_i is strictly better than s_j if the distance computed for s_i is shorter than that of s_j). We then arbitrarily choose one of the remaining entries to be processed next. The same procedure is used for subsequent non-leaf nodes until a leaf node is reached and the tuple inserted. Then, the bounding sets of the parent nodes are modified (either adjusting the ranges in the rangeset or adding the new signature to the set signature) all the way up to the root.

As an example, consider the insertion of a new 3 star hotel, hotel 6, located in the uptown area and having rates of 200. The insertion algorithm first checks the entries in N1. Hotel 6's `area` and `stars` attribute values occur in the respective sets of the bounding set of both entries. However, the distances computed on the rates show that the second entry (on the right) requires the minimum enlargement. As such, the right subtree is chosen. Subsequently, it is inserted in node N121 and the parent nodes are updated to (rates=[190,200], area={uptown}, stars=[3,3]).

We make use of the quadratic-cost R-tree node splitting algorithm whenever an overflow occurs during an insertion. To split a node, we need to split each s_i of the bounding set of the node. The heuristic we adopt is as follows. If s_i is a rangeset, we distribute the ranges evenly between the nodes (making any range adjustments if necessary). If s_i is a set signature, we could compare the signature of each value in the domain with the set signature to check which values belong to s_i before re-distributing the values between the nodes. However, this is computationally expensive, especially when the attribute's domain is large. Instead, we choose to duplicate the set signature and add it to the new node. Obviously, this will lead to more false drops for subsequent queries and may reduce the effectiveness of the Pref-Tree. One possible solution is to rebuild all the set signatures periodically.

As for deletion, we first search for the tuple in the Pref-Tree and delete it from the leaf node. Next, we re-adjust the bounding set in all the parent nodes. The heuristics we adopt are similar to those for insertion. However, following the same argument in RD-tree, we do not restructure the tree to guarantee a minimum

number of entries in each node.

Before we discuss our strategies for query evaluation, we would like to mention that our insertion and deletion heuristics are still preliminary at the moment. We are currently considering other heuristics for more efficient insertion and deletion operations as well as a more thorough analysis of these two operations, but this largely remains as future work.

5.2.3 Evaluation of Pareto Queries

Figure 5.5 shows the algorithmic description of our approach. The algorithm continuously picks a candidate tuple t from the dataset D and then traverses the Pref-Tree having the root node r to see if any tuples dominates t. If there is none, then t is output. The key method in the algorithm is function `isDominated` (also illustrated in Figure 5.5). It is essentially a backtracking algorithm to search the Pref-Tree. We shall briefly discuss the routines it uses. `isLeafNode(n)` returns true if n is a leaf node, false otherwise. `dominates(`x`, `t`, `P`)` returns true if tuple x dominates tuple t based on preferences P, false otherwise.

`canDominate(`t`, `e`, `P`)` (also illustrated in Figure 5.5) is used to drive the search. It returns true if some tuples covered by the bounding set of an entry can potentially dominate t. It relies on two important subroutines. The first is `strictlyDominates(`$e.s_i$`, `t_i`, `p_i`)` which uses the set s_i of entry e to determine whether some tuples covered by e have values that are *strictly better* than t_i for A_i. The heuristics used depend on the preferences under consideration. We shall now illustrate some of these heuristics using examples (full details can be found in Appendix B).

Consider the preference AROUND(A_i, 50). Let t_i be the value of attribute A_i of the candidate tuple. Assuming that the respective rangeset in e is given by [5,10] and $t_i = 20$. From the rangeset, we can see that the shortest distance any tuples covered by e has distance 40 to the desired value whereas the candidate tuple's distance is 30. Hence, the routine returns false as no tuples covered by e has a

value for A_i that can result in a better distance. As another example, consider the preference POS(A_i, $\{x\}$). We first check whether t_i is in the POS-set. If it is, then no tuples covered by e can have a value strictly better than t_i for A_i and false is returned. Otherwise, we check whether x exists in the respective set signature. If the value exists, true is returned. Due to the presence of false drops, x may be falsely deduced to be in the set signature. Consequently, this will result in more searching but the correctness of our algorithm is not affected.

```
Algorithm Pref-Tree
Input: Dataset D, Root node r, Preferences P
1.  foreach t in D
2.      if not isDominated(t, r, P)
3.          output t

Function isDominated(t, n, P)
Input: Candidate tuple t, Pref-Tree node n, Preferences P
1.  if isLeafNode(n)
2.      foreach tuple x in n.data
3.          if dominates(x, t, P)
4.              return true
5.      return false
6.  foreach entry e_i in n.entries
7.      if canDominate(t, e_i, P)
8.          if isDominated(t, e_i.child, P)
9.              return true
10. return false

Function canDominate(t, e, P)
Input: Tuple t, Node's entry e, Preferences P
1.  hasBetter = false
2.  foreach p_i ∈ P
3.      if strictlyDominates(e.s_i, t_i, p_i)
4.          hasBetter = true
5.      else if not valueExist(e.s_i, t_i)
6.          return false
7.  return hasBetter
```

Figure 5.5: Pref-Tree algorithm.

The second routine is `valueExist(`$e.s_i$`,` t_i`)` which simply determines whether t_i exists in one of the ranges in the rangeset or in the set signature, depending on the type of attribute. Again, false drops might cause the routine to claim that

a value exists when it does not. This also does not affect the correctness of our algorithm but could result in more traversals.

We now describe the function `isDominated`. Steps 6-10 handle the case when a non-leaf node is encountered. For each entry in the node, it checks whether there is any possibility of t being dominated by some tuples. If there is such a possibility, the child node is retrieved and checked (steps 7-8). Eventually, if a leaf node is reached, the actual tuples are retrieved and checked against t to see if any of them dominates it (steps 1-5). The following shows an important property of our Pref-Tree algorithm.

Theorem 5.3. *The Pref-Tree algorithm correctly retrieves the desired results over a dataset of size n and requires $O(n^2)$ time in the worst case.*

Proof. Given a candidate tuple t, the algorithm traverses the Pref-Tree to determine whether any tuples dominates t. In the worst case, each and every branch of the Pref-Tree is traversed. This is equivalent to comparing t against all tuples in the dataset. Hence, if some tuples dominate t, t will be eliminated. Otherwise, t will eventually be compared against all tuples before being output, ensuring that the desired results are returned. Moreover, our algorithm takes the conservative approach to ensure correctness by ensuring that extra branches are searched whenever the presence of false drops might affect the correctness of the results. Lastly, in the worst case, all tuples in the dataset are not dominated by any other tuple. Hence, n^2 tuple comparisons are required, resulting in $O(n^2)$ time. □

We now illustrate how the Pref-Tree structure shown in Figure 5.4 is used for answering the sample pareto query on the hotel relation in Table 5.1.

Example 5.4. Consider our running example's query: AROUND(`rates`, 200) ⊗ HIGHEST(`stars`) ⊗ POS(`area`, {uptown}). Assume that hotel 1 is currently the candidate tuple. We start with the root node's first entry (on the left). Since the bounding set indicates that there could be a hotel with better rates, stars and area i.e. some tuples it covers can dominate hotel 1, we traverse the left subtree and

examines the first entry of node N11. Again, it indicates that there is a possibility that some tuples it covers can dominate hotel 1 and so we move to node N111. Since this is a leaf node, we retrieve the data tuples and compare them with hotel 1. Since hotel 5 dominates hotel 1, the search is terminated. Now, let us consider hotel 4. Similarly, we start with the first entry of node N1 and then examine the first entry of node N11. However, since the rangeset of its `stars` attribute is [2,2] while that of hotel 4 is 4. There is no way any tuples covered by this entry can dominate hotel 4. Thus, we skip that branch and check the second entry. Since the second entry indicates that some of its tuples might dominate hotel 4, we check the first entry of node N112. As this is a leaf node, we retrieve the data tuples and compare with hotel 4. Since no tuples can dominate hotel 4, we check the second entry which gives us the same result and we then backtrack to the second entry of the root node. In this entry, the rangeset of the `stars` attribute is strictly lower than that of hotel 4. Hence, there is no way any tuples covered by the entry can dominate hotel 4 and hence hotel 4 is output as an answer.

5.3 A B-tree-based Approach

The previous two approaches each utilizes a multi-dimensional index structure for processing pareto queries. In this section, we introduce our third approach which is based only on *single-dimensional indexes* e.g. B^+-trees. While it requires a single dimensional index to be built for each attribute that may be queried, these indexes are still comparably cheaper to maintain than a single multi-dimensional index. Moreover, since most commercial DBMS support at least one type of single-dimensional index, this approach is easy to integrate into existing database systems.

5.3.1 The Model

Given the dataset D with d attributes $A_1, \ldots, A_d$, assume that each tuple $t_i \in D$, for $1 \leq i \leq |D|$, is given a unique tuple id (tid) which is equal to i. We shall also

assume that for each attribute A_j, a single-dimensional index, $index_j$, is created to index attribute A_j's values. While our approach does not depend on the type of index used, we adopted the B$^+$-tree index in our work as it is available in all commercial DBMS. From the index of A_j, we can derive a list of entries of the form $<tid, value>$ where tid is the tuple id of some tuple $t_i \in D$ and $value$ is t_i's value for A_j (denoted by $t_i.A_j$). Each list of entries has the following properties:

Property 5.1. *For each tuple $t_i \in D$, $1 \leq i \leq |D|$, there exists one and only one entry with content $<i, t_i.A_j>$ in list j, $1 \leq j \leq d$. Thus, there are $|D|$ entries in each list.*

Property 5.2. *Assume that a base preference P_j is specified on attribute A_j. Let L_p denotes a sublist of entries that are derived from the index of A_j such that each entry in L_p has the same entry value p. For any two sublists of entries, L_p and L_q, where $p \neq q$, we write $L_p \approx L_q$ to mean that the values of the entries in L_p and L_q are incomparable with respect to P_j and either L_p can be derived first or L_q. On the other hand, we write $L_p < L_q$ if the value of the entries in L_q is more preferred i.e. strictly better than those in L_p. In this case, L_q must be derived before L_p. Now, if A_j has k distinct values $(v_1, \ldots, v_k)$, the list of entries is derived from the index of A_j in such a way that*

$$L_{v_1} \approx \ldots \approx L_{v_g} < \ldots < L_{v_h} \approx \ldots \approx L_{v_i} < L_{v_j} \approx \ldots \approx L_{v_k}$$

where values of entries in L_{v_j} to L_{v_k} are incomparable to each other but all of them are more preferred than the next set of incomparable values in the entries starting from L_{v_h} to L_{v_i} and so on. We call such an order the ***maximal order*** [1].

It is relatively easy to derive the entries in the maximal order when the underlying index is the B$^+$-tree. For example, if a HIGHEST preference is specified on an attribute, we just need to scan the leaf nodes of the B$^+$-tree in non-ascending order of the attribute values to create the entries in maximal order. Note that these entries are not pre-computed but are created on the fly during query evalu-

[1] We note that the maximal order is also an instance of a weak order. Deriving objects in non-increasing weak order is also known as sorted access [38, 48, 49].

ation. Since our approach only requires that the entries form a maximal order but is independent of how they are derived from the indexes, we shall assume for now that entries derived from the indexes always form a maximal order. Towards the end of this section, we describe how this is achieved using the B^+-tree indexes for each base preference.

Example 5.5. Consider our hotel relation in Table 5.1. Assume that an index is constructed for the three attributes `rates`, `area` and `stars`. Figure 5.6 shows the list of entries derived from the indexes based on the preferences specified in our sample pareto query. For the `stars` attribute, the entries are ordered in non-ascending order of the attribute values. For the `rates` attribute, the ordering is based on the difference between a hotel's rate and the ideal rate of 200 e.g. hotel 2's rate is only \$200 - \$190 = \$10 lesser than the ideal \$200. This is followed by hotel 5, with the second smallest difference of \$57 and so on. In other words, entries whose values have the smallest differences are derived first. For the `area` attribute, we can see that hotels in the favored area (uptown) are derived first. As an example of the case where derived entries are incomparable, consider the base preference specified on `area` to be POS(`area`,{uptown, midtown}) instead. In this case, it does not matter whether entries for hotels 2 and 5 or entries for hotels 1, 3 and 4 are derived first, although entries having the same value should be derived consecutively e.g. deriving hotel 2, followed by hotel 3, then hotel 5 is not allowed in this case.

tid	Rates
2	190
5	257
1	280
3	308
4	314

AROUND(200)

tid	Area
2	uptown
5	uptown
1	midtown
3	midtown
4	midtown

POS(uptown)

tid	Stars
4	4
2	3
3	3
1	2
5	2

HIGHEST

Figure 5.6: List of entries for the running example.

5.3.2 The Pareto Algorithm

We shall now present our algorithm for evaluating pareto preference queries, assuming that entries are derived from the indexes in maximal order. WLOG, we assume that the query consists of d preferences, one for each attribute of D. For convenience, we shall call the set of answer tuples from evaluating the pareto preference query to be **maximal tuples**.

Data Structures

We shall first describe the auxiliary structures used in our algorithm. The first structure is a bitmap containing a single array of $|D|$ bits. The ith bit of the bitmap represents tuple t_i and is set to 1 when t_i is either found to be a maximal tuple or is found to be dominated by a maximal tuple. The bitmap is used to determine when the algorithm can terminate as well as to reduce the search space in each iteration.

The second structure is a hash table that keeps a record of the number of times a *tid* is "seen". Whenever the algorithm examines an entry derived from any of the indexes during evaluation, the *tid* of that entry is considered "seen" and the number of times this *tid* is seen is incremented by one in the hash table. The hash table is used for detecting a **match**. A match occurs when the number of times a *tid* is seen is equal to the number of preferences specified. The tuple whose *tid* results in a match, called the *matched tuple*, represents a potential maximal tuple.

Lastly, for each index, $index_j$, we store a bitmap structure, $seen_j$, consisting of $|D|$ bits for that index. Similar to the first structure, the ith bit of $seen_j$ is used to represent tuple t_i. However, it is set to 1 only when the entry containing the *tid* of t_i is derived from $index_j$. Its purpose is to support the updating of the first bitmap structure.

The Main Algorithm

Figure 5.7 gives the algorithmic description of our approach. The first bitmap structure described in the previous subsection is denoted by *bitmap* while the hash table is denoted by *htable* in the algorithm. After initializing, the algorithm proceeds in a number of iterations. Each iteration adopts a **search-and-reduce** strategy which is made up of two phases. The first phase is used to *search* for some maximal tuples to output while the second phase uses the maximal tuples found in the first phase to *reduce* the search space for subsequent iterations by updating *bitmap*. The algorithm terminates when all bits in *bitmap* are set to 1 which indicates that each tuple in D is either output (because it is a maximal tuple) or eliminated (by some maximal tuples). Notice that the algorithm is **progressive** as only a subset of the maximal tuples are output in each iteration. We now describe the 2 key phases.

Algorithm Pareto
Input: Preferences $P = (P_1, \ldots, P_d)$
1. // initialization
2. set *bitmap* to 0 and *htable* to empty
3. **while** number of 1s bits in *bitmap* $\neq |D|$
4. // phase 1
5. $results \leftarrow$ findMaximal(*bitmap*, *htable*, P)
6. // output results
7. **foreach** $t \in results$, output t as an answer
8. // phase 2
9. updateBitmap(*bitmap*, $results$, P)

Figure 5.7: The main algorithm.

Phase 1: Searching for maximal tuples

The approach we adopt for finding maximal tuples in this phase is similar that proposed in [6, 49]. However, as the context and overall strategy of our work is different from them, similar results that are repeated here will be presented within the context of our framework.

Figure 5.8 depicts the details of function `findMaximal`. The function consists of two main steps. The first step (lines 3-11) is an adaptation of the first step of

Fagin's A_0 algorithm [36]. In this step, entries are derived *simultaneously* from the indexes for processing i.e. the first entry from each index is derived first followed by the second entry from each index and so on. Routine `getNextEntry` is responsible for deriving the entries from the indexes in maximal order (line 6). It uses *bitmap* to decide whether an entry can be discarded i.e. when the corresponding bit for the entry's *tid* is set to 1 in *bitmap*, or returned to `findMaximal` for processing. $seen_j$ and v_j are then updated with the returned entry's *tid* and *value* respectively (line 7). Function `updateHTable` is called next to update *htable* (line 8). After every update, the function returns a boolean flag to indicate whether a match has occurred. If a match occurs, the matched tuple, $t_{e.tid}$, is retrieved for inclusion into the list of results, *anslist*, before exiting the first step (lines 9-11).

Function findMaximal
Input: Bitmap *bitmap*, Hash table *htable*
Preferences $P = (P_1, \ldots, P_d)$
Output: A set of maximal tuples
1. $anslist \leftarrow \emptyset$; $match \leftarrow false$
2. // Step 1
3. **while not** *match*
4. **for** $j = 1$ to d
5. **if** $index_j$ has entries to derive **then**
6. $e \leftarrow$ getNextEntry(*bitmap*, $index_j$, P_j)
7. $seen_j$.set($e.tid$); $v_j \leftarrow e.value$
8. $match \leftarrow$ updateHTable(*htable*, $e.tid$)
9. **if** *match* **then**
10. $anslist \leftarrow anslist \cup t_{e.tid}$
11. **break**
12. // Step 2
13. **for** $j = 1$ to d
14. **while** value of next entry from $index_j = v_j$
15. $e \leftarrow$ getNextEntry(*bitmap*, $index_j$, P_j)
16. $seen_j$.set($e.tid$)
17. $match \leftarrow$ updateHTable(*htable*, $e.tid$)
18. **if** *match* **then** $anslist \leftarrow anslist \cup t_{e.tid}$
19. remove dominated tuples from *anslist*
20. **return** *anslist*

Figure 5.8: Function findMaximal.

Definition 5.2 (Matched tuple). Let L_j denote the list of entries derived from the index of attribute A_j, for $1 \leq j \leq d$. A matched tuple is a tuple in D whose *tid* is seen in each L_j when the entries are derived from the indexes simultaneously. Since each entry is derived only once in this phase, the *tid* is seen exactly d times.

In step 2 (lines 13-18), the processing is similar to step 1 except that only entries with values v_j are derived from $index_j$ for processing. After step 2 is completed, any tuples in *anslist* that are dominated by some other tuples in the same list are removed (line 19). This ensures that the final tuples returned (line 20) do not dominate each other.

We shall now show that the tuples retrieved by `findMaximal` in the **first** iteration of the algorithm are a subset of the maximal tuples of the query. Subsequent iterations will be discussed after we have described the second phase. For a start, we would like to show the rationale for the second step since the first step alone seems sufficient to determine maximal tuples. An example would make this clear. Figure 5.9(a) shows a dataset having 2 attributes and 3 data tuples. Assume that the query consists of HIGHEST preferences specified on both attributes. We show the entries derived from the indexes in Figure 5.9(b). Notice that the first match occurs at *tid* 2. However, tuple 2 is not a maximal tuple as it is dominated by tuple 1. We note, that the first step is sufficient if each tuple in the dataset has distinct attribute values from each other, but such cases are uncommon in practice.

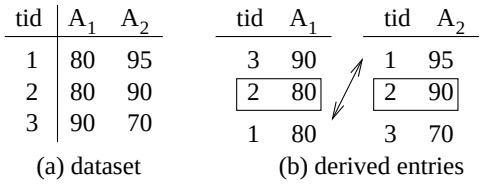


Figure 5.9: Rationale for the second step.

Let us place the above scenario into perspective. Let t_p be the matched tuple found in step 1 and t_q be a tuple whose *tid* is seen in step 1 but does not result in a match. This means that t_q has at least one attribute value, say $t_q.A_j$, that might be 1) strictly worse than $t_p.A_j$ or 2) incomparable to $t_p.A_j$ or 3) equal to $t_p.A_j$. In

the first two cases, there are no way t_q can dominate t_p but such a possibility exists in the third case. The purpose of the second step is thus to retrieve tuples that fall in the third case. Before we show that the tuples returned by `findMaximal` are indeed maximal tuples of the query, we first state an important lemma:

Lemma 5.1. *After the execution of* ***findMaximal****, a tuple whose tid is not seen cannot dominate the tuples in anslist.*

Proof (By contradiction). Let t_p be a tuple whose *tid* is not seen after the execution of `findMaximal` but dominates at least one tuple, t_q, in *anslist*. To dominate t_q, t_p must have at least one of its attribute values, say $t_p.A_j$, strictly better than $t_q.A_j$. However, since entries are derived in maximal order from the indexes, this means that the *tid* of t_p has to be seen during the execution of `findMaximal`. Therefore, t_p cannot dominate any tuples in *anslist*. □

Lemma 5.2. *Only maximal tuples are returned by function* ***findMaximal*** *in anslist.*

Proof (By contradiction). Assume that there exists a tuple $t_p \in$ *anslist* that is not a maximal tuple. This means that there must exist some tuple t_q that dominates t_p. Due to Lemma 5.1, the *tid* of t_q must be seen. Now, if the *tid* of t_q results in a match in either step 1 or 2, t_q would have been retrieved and compared with tuples in *anslist*, causing t_p to be eliminated. Therefore, the *tid* of t_q is seen but does not result in a match. This also means that at least one of the attribute values of t_q is either incomparable or strictly worse than the corresponding attribute value of the tuples in *anslist*. In either case, t_q cannot dominate t_p. Thus, t_q cannot exist and we can conclude that the tuples in *anslist* are maximal tuples. □

Example 5.6. We shall now illustrate the first phase of the first iteration of our Pareto algorithm using our running example (see Figure 5.10). In the first step of `findMaximal`, entries are derived from the indexes simultaneously. The first match occurs on *tid* 2. Hotel 2 is thus retrieved and stored in *anslist*. In the second step, an additional entry is derived from the index of `stars` as it has the same value as its last derived entry i.e. 3. Since there is no further matches, hotel 2

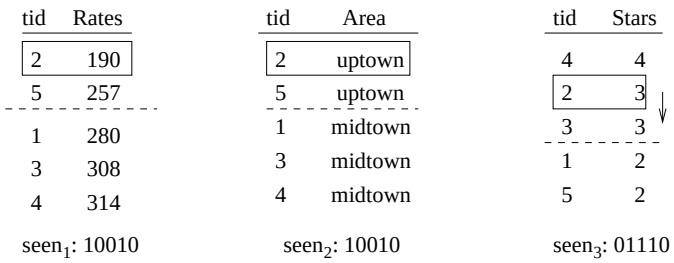


Figure 5.10: Illustration of findMaximal.

is returned and subsequently output as an answer. On the same figure, we also show the current content of $seen_j$ where $j = 1$, 2, and 3 represents the content for attributes `rates`, `area` and `stars` respectively. The dashed lines separate entries that have been examined from those that have not.

Phase 2: Reducing the search space

The main purpose of the second phase is to use the maximal tuples found in the first phase to identify those tuples that are dominated by them. Since these dominated tuples cannot be part of the answers, we want to exclude them from subsequent processing. Moreover, we would also want to exclude the maximal tuples found in the first phase from subsequent iterations as they are incomparable to those tuples that they cannot dominate. These exclusions essentially reduce the search space for later iterations. We shall now present an important lemma that will help us identify those tuples we want to exclude:

Lemma 5.3. *Let x be a maximal tuple where $x.A_j$ denotes the value of attribute A_j, for $1 \leq j \leq d$. If entries are derived from each index up to but excluding the entry containing $x.A_j$, then tuples whose tids are currently seen cannot possibly be dominated by x. Furthermore, if we exclude from the set of unseen tids, those tids of tuples that have incomparable attribute values as $x.A_j$, then a tid that still remains unseen is either dominated by x or has the same values as x.*

Proof. Because entries are derived in maximal order, a tuple whose *tid* is seen will have at least one of its attribute values either strictly better or incomparable to the corresponding attribute value of x. In either case, x cannot dominate the tuple.

On the other hand, if we exclude from the unseen *tids*, those *tids* of tuples that have attribute values that are incomparable to x's, the remaining set of unseen *tids* are *tids* of tuples that have attribute values that are equal or strictly worse than x's. Hence, they are either dominated by x or have the same values as x. □

The reduction of the search space is achieved by manipulating the global bitmap structure, *bitmap*. Recall that in phase 1, as entries are derived from the indexes, *bitmap* is used to determine whether a derived entry needs to be processed or it can be skipped. Hence, by indicating on *bitmap* those tuples that can be excluded from subsequent evaluation, the entries of these tuples will be skipped in the first phase of subsequent iterations, making the search in phase 1 more efficient.

```
Procedure updateBitmap
Input: Bitmap bitmap, L = {t_i} for 1 ≤ i ≤ |L|,
       Preferences P = (P_1, ..., P_d)
1.  foreach tuple x ∈ L
2.      seenset ← 0
3.      for j = 1 to d
4.          temp_j ← seen_j // make a copy of seen_j
5.          end ← current position of index_j
6.          reset index_j such that next derived entry
                starts from the first entry with value x.A_j
7.          do
8.              e ← getNextEntry(bitmap, index_j, P_j)
9.              temp_j.reset(e.tid) // 'unsee' the tid
10.             pos ← current position of index_j
11.         while pos ≠ end
12.         addUnrank(index_j, temp_j, x.A_j, P_j)
13.         seenset = seenset | temp_j
14.     // update bitmap
15.     bitmap ← bitmap | ! seenset
16. // update seen_j using the updated bitmap
17. b ← ! bitmap
18. for j = 1 to d
19.     seen_j ← seen_j & b
```

Figure 5.11: Procedure updateBitmap.

Phase 2 is realized in routine `updateBitmap`. Figure 5.11 shows the algorithmic description of this routine. The list L contains the maximal tuples found in phase 1 of the current iteration. For each maximal tuple, x, the algorithm re-derives en-

tries starting from the one with value $x.A_j$ until the last derived entry examined in phase 1 (lines 4-11). As entries are re-derived, a local copy of $seen_j$, $temp_j$, is updated by setting the respective bits for the *tids* of the entries to 0. This is equivalent to deriving entries from the index up to but excluding the entry containing value $x.A_j$. Subsequently, $temp_j$ is updated by routine `addUnrank` to include the *tids* of tuples having attribute values that are *incomparable* to $x.A_j$ (line 12). It is then bitwise `OR` with *seenset* (line 13). *seenset*, thus, captures the *tids* of tuples that either have strictly better or incomparable values to $x.A_j$. Eventually, by executing a bitwise *not* operation on *seenset* (line 15), *seenset* now indicates *tids* of tuples that have the same attribute values as x or are dominated by x (by Lemma 5.3). Executing a bitwise `OR` with *bitmap* subsequently will update *bitmap* to include the *tids* of tuples having the same attribute values as x as well as tuples x dominates.

Once all the maximal tuples in L have been processed, the updated *bitmap* is used to update the individual $seen_j$. This is necessary because if a *tid* is now excluded from subsequent processing (as indicated in *bitmap*), it should not be considered during the processing of $seen_j$ i.e. the corresponding bit in $seen_j$ should be set to 0. This step can be carried out efficiently by executing a bitwise *and* operation between $seen_j$ and a copy of the negation of *bitmap* (lines 17-19). This completes the second phase.

Example 5.7. Refer to Figure 5.12. The maximal tuple found in the first phase is hotel 2 (190, uptown, 3). First, `updateBitmap` will re-start from the first entries containing the values 190, uptown and 3 for attributes `rates`, `area` and `stars` respectively (entries below the bold line in Figure 5.12). As it derives the entries from each index, a duplicate copy of $seen_j$, $temp_j$, is updated and eventually bitwise `OR` with *seenset*. The final state of $temp_i$ for each attribute is shown in the figure as well. Finally, by executing a bitwise `NOT` on *seenset* followed by a bitwise `OR` with *bitmap*, *bitmap* is now updated to reflect the *tids* of tuples that can be ignored in subsequent iterations. Specifically, these are tuples 1, 2, 3 and 5. Tuple 2 is the

original maximal tuple while tuples 1, 3 and 5 are dominated by tuple 2. The final content of *seenset* and *bitmap* are also shown in Figure 5.12.

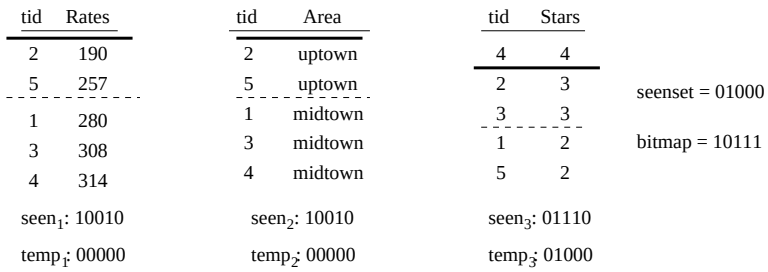


Figure 5.12: Illustration of updateBitmap.

Notice that procedure `updateBitmap` is called for *each* maximal tuple found in the current iteration. This could be computationally expensive if the current iteration finds many maximal tuples. One possible solution might be to do the "unseening" process once using the best attribute value for each attribute of the maximal tuples. Such a solution, however, only works for the case when the query consists of 2 preferences. Consider Figure 5.13, which shows 3 sets of derived entries. Assume that the query consists of 3 HIGHEST preferences. The first match occurs for *tid* 1. Next, based on the value of the last entry derived for each index, an additional entry is derived for index 3 i.e. (2, 60). Consequently, two maximal tuples are found (tuples 1 and 2). If we now use the best attribute value for the "unseening" process i.e. (90, 80, 70), *tid* 3 would be considered unseen and eliminated from subsequent consideration. However, neither tuple 1 nor tuple 2 dominates tuple 3.

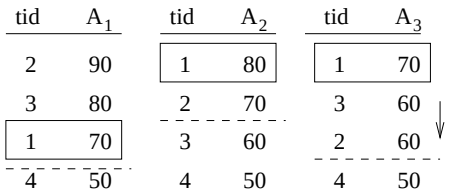


Figure 5.13: Evaluation of query with 3 preferences.

We now discuss an approach which works relatively well in our experimental study in solving the aforementioned problem. It works in a similar way as what we have just described, using the best attribute values among all the maximal tuples.

To overcome the problem when the query consists of more than 2 preferences, after we completed the "unseening" process using the new attribute values, for each maximal tuple, we derive entries from each indexes until we reach the same point if we were to do the "unseening" process for just that tuple. After this, we "unsee" the *tids* just seen for the tuple in the usual way. The rest of the maximal tuples are then processed in a similar way.

We would also like to point out a subtle point regarding incomparable values. Recall that in Figure 5.11, routine `addUnrank` indicates on $temp_j$ the *tids* of tuples having values that are incomparable to $x.A_j$. If $x.A_j$ has many incomparable values, this step becomes computationally expensive. Such a scenario always occurs when preferences are specified on unordered attributes. For example, consider a NEG preference specifying a single disliked value v for some attribute A_j. If a maximal tuple having value $w \neq v$ for A_j is found, `addUnrank` has to look for *tids* of tuples which do not have values w or v for A_j. This set of *tids* is usually very large unless the dataset contains many tuples having values w or v for A_j. This makes `addUnrank` a very expensive operation. However, we can exploit the fact that preferences specified on unordered attributes usually have very few levels in their better than graphs.

We now show how this problem is solved for the NEG preference discussed earlier. Other types of related preferences can be solved in a similar way. First, prior to evaluating the query, we find the *tids* of tuples having value v first and set on a separate bitmap, *pre*, the bits corresponding to these *tids*. Subsequently, when we need to find the incomparable values of w, we first execute a bitwise `NOT` on a copy of *pre*. Next, we look for *tids* of tuples having values w and set their corresponding bits in *pre* to 0. Therefore, *tids* of tuples having values incomparable to w will be marked in that copy of *pre* and we can use it to update $temp_j$. From our experimental study, we found that this approach is very effective in handling preferences that are specified on unordered attributes.

Subsequent iterations of the algorithm are executed in a similar manner by

repeating both phases. However, as a result of updating *bitmap* in phase 2 of each iteration, phase 1 of the next iteration will have fewer entries to process. This makes the overall processing more efficient.

Example 5.8. A simple illustration of a subsequent iteration is shown in Figure 5.14. We show the updated $seen_j$ as well as *bitmap* in the figure. Entries are

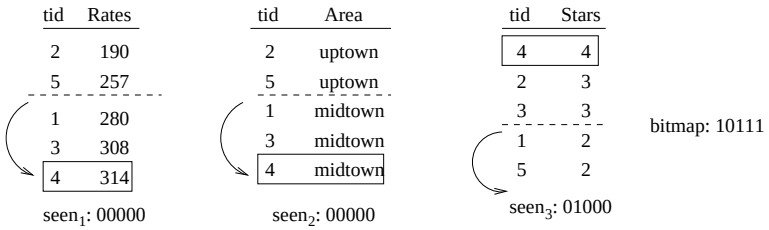


Figure 5.14: Subsequent iteration.

derived starting from the ones after the dashed lines. Since *bitmap* indicates that tuples 1, 2, 3 and 5 should be ignored, entries for *tid* 1 is skipped for all attributes and entries for *tid* 3 is skipped for attributes `rates` and `area` while entry for *tid* 5 is skipped for attribute `stars`. A match occurs on *tid* 4 in this iteration and is determined to be a maximal tuple. At the end of this iteration, *bitmap* will contain only 1s bits and the algorithm terminates.

Analysis of the Pareto Algorithm

Theorem 5.4 (Correctness and Completeness). *The algorithm always terminates and outputs all and only the maximal tuples of the pareto preference query.*

Proof. We first prove by induction that the algorithm outputs only maximal tuples in *each* iteration.

Basis Step: We have shown in Lemma 5.2 that only maximal tuples are output in the first iteration.

Induction Hypothesis: Assume that the Pareto algorithm outputs only maximal tuples in each iteration up to the nth iteration. Thus, at the end of this iteration, *bitmap* would indicate the *tids* of maximal tuples found in this and earlier iterations as well as those tuples dominated by them.

Induction Step: Consider the $(n+1)$th iteration. Let t_p be a tuple that is dominated by a maximal tuple t_q. Clearly, at least one attribute value of t_p must be strictly worse than the corresponding attribute value of t_q while the rest of the attribute values are either equal or strictly worse than the corresponding attribute values of t_q. We need to show that in this iteration, t_p is not output if A) t_q is found in an earlier iteration; B) t_q is to be found in the current iteration; C) t_q is to be found in a later iteration.

First, if t_q is found in an earlier iteration, by the induction hypothesis, the *tid* of t_p would be reflected in *bitmap*, causing its entries to be ignored in the current iteration. Hence, t_p can never be output if t_q is found in an earlier iteration.

Consider case B. For t_p to have any chance of being output in this iteration, a match has to occur for t_p. Assuming that a match does occur for t_p. Since t_q is also found in the current iteration, it will eventually be compared to t_p and found to dominate t_p. Hence, t_p can never be output if t_q is found in the current iteration.

Now, consider case C. Since each attribute value of t_p is either equal or strictly worse than the corresponding attribute value of t_q, a match cannot possibly occur for t_p without occurring for t_q as well. As t_q is to be found in a later iteration, t_p can never be output in the current iteration. This completes the proof and we can conclude that our Pareto algorithm only outputs maximal tuples in each iteration.

Our Pareto algorithm always terminates because at the end of each iteration, *bitmap* is always updated to include the *tids* of the maximal tuples found in the current iteration and those tuples they dominate. Those tuples having *tids* whose corresponding bits in *bitmap* are not set will have at least one attribute value which is incomparable or strictly better than the maximal tuples found. Subsequently, they are not ignored during evaluation and will result in a match or found to be eliminated, causing their corresponding bits to be set to 1. Hence, all bits in *bitmap* would be eventually set to 1 after which the algorithm terminates.

Finally, the algorithm never misses a maximal tuple. Consider a maximal tuple, t_p, that is not yet determined to be a maximal tuple in the current iteration.

Clearly, for each maximal tuple t_q found in the current iteration, at least one of the attribute values of t_p must be incomparable or strictly better than that of t_q's. Consequently, its *tid* will not be indicated in *bitmap* at the end of this iteration. Therefore, in subsequent iterations, entries containing its *tids* will be processed and a match will occur for t_p which is then determined to be a maximal tuple. □

Theorem 5.5 (Optimality). *The Pareto algorithm retrieves and processes an optimal number of data tuples in each iteration.*

Proof. Data tuples are only retrieved in the first phase of the Pareto Algorithm. We have shown that the matched tuple retrieved in step 1 is not guaranteed to be one of the answers and more tuples may have to be retrieved in step 2. By Lemma 5.2, at least one of the tuples retrieved in either one of these steps will be a maximal tuple. Moreover, each retrieved tuple (whether it is a maximal tuple or dominated by a maximal tuple) would be marked in *bitmap* and thus will not be retrieved again in subsequent iterations. This means that the number of tuples retrieved in phase 1 of an iteration has to be the minimum and hence the optimal number of tuples that has to be retrieved. □

Theorem 5.6 (No redundancy). *Each maximal tuple of the pareto preference query is output only once.*

Proof. After each maximal tuple is output, its *tid* will be indicated in *bitmap*. Subsequently, its *tid* is always ignored during evaluation and will never result in a match again. Therefore, each maximal tuple will only be output once. □

Theorem 5.7. *The Pareto algorithm requires $O(n^2)$ time in the worst case for a dataset of size n.*

Proof. In the worst case, all tuples in the dataset are not dominated by any other tuple. Hence, all tuples will eventually be compared with each other i.e. n^2 comparisons are required, resulting in $O(n^2)$ time. □

Deriving Entries for the Base Preferences

We now discuss how to derive entries in maximal order from the B^+-tree indexes for the base preferences:

Base preferences on ordered attributes. First, for the AROUND preference, we start with the index entry in the B^+-tree that has a value that is closest to the desired value (in the case of a tie, we arbitrary choose one). To derive the next entry, we examine the neighboring index entries of the previous index entry as they are guaranteed to have the next shortest distance. Deriving entries for the BETWEEN preference is analogous to that for AROUND preference except that the distance is measured from the range instead of a desired value. For a HIGHEST (LOWEST) preference, we examine the B^+-tree index entries in non-ascending (non-descending) order of attribute values.

Base preferences on unordered attributes. For these set of base preferences, we can derive the entries from the indexes in a generic way. We first build a better than graph for the respective preference. Then, we retrieve those index entries having values that are in the highest level of the graph first, followed by those in the second level and so on. For example, if the preference is POS(A_j, $\{v\}$), we start with the B^+-tree index entries having values v followed by those with values not equal to v.

5.4 Performance Study

To evaluate the effectiveness of our proposed algorithms, we conducted an extensive set of experiments to study its performance. All experiments are carried out on a Pentium 4 PC with a 2.4 GHz processor and 256 MB of main memory running the Linux operating system. Besides implementing our proposed algorithms: Bitmap (bitmap), Pref-Tree (ptree) and B-tree (index), we also adapted the Block Nested Loop algorithm (bnl) in [9] to handle pareto preference queries, and implemented a disk-based version of the Best operator proposed in [100] (best). We also extended

the B-tree algorithm proposed in [9] to handle pareto preference queries (btree). The extended B-tree algorithm works as follows. The first part of the algorithm is similar to the first phase of our B-tree approach. However, as the original algorithm does not handle incomparable values, we extended the first phase as follows. Let v_j be the value of the last retrieved entry of $index_j$ when the first match occurs. Then, in the second step, besides deriving entries having the same value as v_j from $index_j$, any derived entry with an incomparable value to v_j will also be processed as well. At the end of this step, any *tids* not seen by now cannot be a maximal tuple. The remaining set of seen *tids* are then used to retrieve the actual data tuples and a block nested loop algorithm applied on them to find the rest of the maximal tuples.

The datasets used in all our experiments are generated in a similar way as described in [9, 97]. Each dataset contains 100000 tuples, each of size 300 bytes. Each tuple has 10 attributes of type *integer* and one "bulk" attribute that is packed with garbage characters to ensure the tuple is 300 bytes long (the "bulk" attribute is ignored during processing). There are 7 ordered attributes and 3 unordered attributes. The domain of ordered attributes is (0, 1000] while the domain of unordered attributes is (0, 200]. Three types of datasets are generated: (1) in the **independent** datasets, attribute values of tuples are generated using a uniform distribution; (2) in a **correlated** dataset, tuples whose attribute values are good in one attribute are also good in other attributes and (3) in an **anti-correlated** dataset, tuples whose values are good in one attribute are bad in one or all of the other attributes. Details on the generation of the datasets can be found in [9].

Test queries consist of either 2, 3 or 4 preferences. The preferences used for ordered attributes are AROUND, BETWEEN, HIGHEST and LOWEST while preferences POS, NEG, POS/NEG and POS/POS are used for unordered attributes. We briefly describe how the preferences are generated. For the AROUND preference, a number in the range of (0, 1000] is picked randomly as the desired value. The BETWEEN preference is similar; a number in the range of (0, 1000] is chosen

as the low value and a value between (0, 50] is randomly generated to indicate the spread. The up value is thus the sum of the low value and the spread. For the non-numerical preferences, the respective set(s) of values for each preference range from 1 to 5 values, with each value in the range of (0, 200].

We generate 100 queries each for queries containing 2, 3 and 4 preferences for each type of dataset. However, we notice that the number of results for each query can vary widely, ranging from a few hundred to a few thousand answers. The number of results can affect the runtime drastically. For example, when there are fewer number of results, bnl is very fast since the desired tuples can all fit in memory. For larger number of results, bnl is extremely slow as more iterations are required. best is also slow for large number of results as it needs to scan the unresolved set once per result. Since we are recording the average timing over a set of queries, we want to avoid the extreme cases. We do this by running the 100 queries first and then extracting those queries whose total number of results fall in a certain range. For queries with 2 and 3 preferences, the range is 100 to 1000 while the range for queries with 4 preferences is 1000 to 10000. The selected queries are then re-run and their average timings computed. We believe that this will result in a fairer comparison between the various algorithms. We also do not test beyond 5 preferences since it is unlikely for a user to give so many preferences in a single query. Moreover, as the number of preferences increases, the number of results increases rapidly. For example, for a pareto query consisting of 5 preferences, the results may be as high as 50000. That is already half the size of the dataset and hence not meaningful in general.

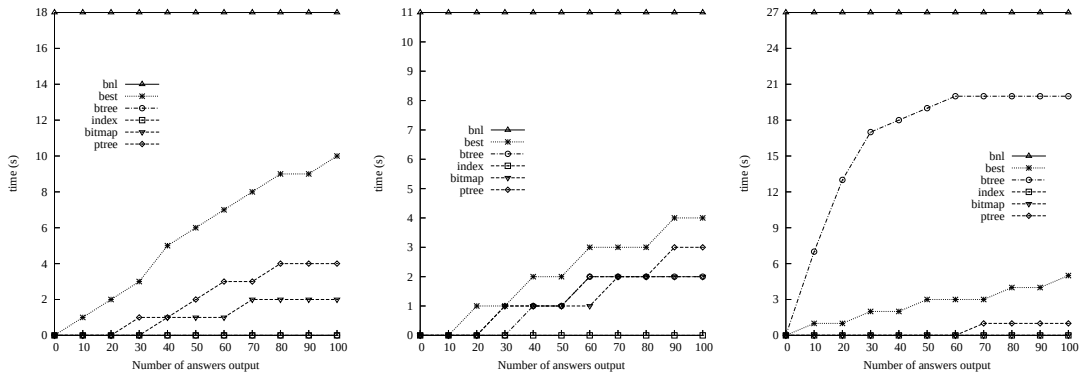
For the B-tree based algorithms (index and btree), one index for each attribute is created prior to running the experiments. For best, a B^+-tree index is used for storing unresolved tuples during query evaluation and we maintain a window size of 1MB for bnl as it is shown in [9] that bnl performs well with this amount of memory. We also conducted a set of sensitivity studies on the parameters to be used for bitmap and ptree prior to running the main experiments. However, due

to space constraints, we are not able to present them here. For bitmap, we use a default of 10 segments of 10000 bits each for each attribute value. For ptree, the default number of ranges in a rangeset is set to 5 while the size of the signature is 256 bits and 3 bits are used to represent each value. The default page size is 4K and the resulting node size is 432 bytes.

We conducted the experiments on various types of datasets. However, since the relative performance of the algorithms are similar, we will only present one representative set of results which is based on independent datasets.

5.4.1 Initial Response Time

In this experiment, we examine the performance of the various algorithms in returning first few answers quickly. We recorded the time each algorithm took to output every 10 answers, up to 100 answers. Returning initial answers quickly is important since most users will generally be interested in the top few results that are returned first. We tested the algorithms using 2, 3 and 4 preferences. Figure 5.15 shows the results for independent datasets.



(a) 2 preferences (b) 3 preferences (c) 4 preferences

Figure 5.15: First 100 results, independent datasets.

From Figure 5.15, we can see that all our proposed algorithms are able to return the first few answers very quickly. In fact, the first answer is almost instantaneous in most experiments. Furthermore, all three algorithms' performance in terms of returning first answers are mildly affected by the number of preferences specified in

the queries. Among the three proposed algorithms, index has the best performance in terms of returning first answers. The key contributing factor is the existence of preferences on unordered attributes. Recall that in index, after a match occurs when scanning the indexes simultaneously, each index is further scanned for entries containing the same values as the match's. Since there are fewer distinct values for unordered attributes, this results in more entries being derived for them. Moreover, most of these entries are pointing to actual answer tuples. For example, for a POS preference, the entries containing the desired values (specified in the preference) are derived first. Since no other tuples can possibly dominate the tuples referenced by these entries, this means that these entries are actually pointing to actual answer tuples. Subsequently, the first phase alone is sufficient to produce the first 100 answers. This is why index can, in general, produce the initial answers very quickly.

For bitmap and ptree, their performance in terms of producing first answers are fairly close, with ptree equal or slower than bitmap. The slower response observed in ptree is largely due to processing queries involving NEG preferences. For NEG preferences, if a candidate tuple's attribute value is in the NEG-set, the approach will conservatively assume at each node that there is a possibility that some tuples covered by the node have respective attribute values that are strictly better than the candidate tuple's value. This generally results in more branches to be traversed, affecting ptree's overall performance. An interesting observation is the 'staircase' pattern that is occasionally observed for both algorithms. This is because the performance of both algorithms are affected by the clustering of the dataset. If a set of tuples that are accessed consecutively from the dataset are all answer tuples to the query, the time interval between the output of these answers will be hardly noticeable. Such a case arise prominently for Figures 5.15(a) and 5.15(b) where there are several small batches of consecutive answers.

For btree, its performance is good when few preferences are specified. This is due to the way btree handles incomparable values. If the first match has an attribute value that has many incomparable values, a lot of entries need to be

derived and processed after the match occurs. Consequently, with more preferences, the overheads add up, resulting in a poorer performance. For best, it is also able to produce the first set of answers fairly fast but does take some substantial amount of time to generate all the 100 results. Recall that best keeps a set of unresolved tuples at the end of each iteration. The unresolved set is large if results with low *selectivity* is encountered as few tuples are eliminated. Thus, the next iteration will be slow since a large number of tuples need to be examined again. This explains why it cannot generate first answers very quickly. Notice that best's ability to return first answers is not severely affected by the increase in the number of preferences in the queries. However, it does occasionally exhibit the 'staircase' pattern. In best, if an initial selected tuple is not an answer tuple, it would be replaced during evaluation, the replacing tuple has to be later compared with tuples in the unresolved set that have yet to be compared with it. This results in a longer time to determine an answer. On the other hand, if an initial selected tuple is an answer tuple or one that is quickly replaced with an actual answer tuple, a shorter time is needed to output the answer. Therefore, the 'staircase' pattern results if there exists batches of answer tuples or tuples that are quickly replaced with actual answer tuples.

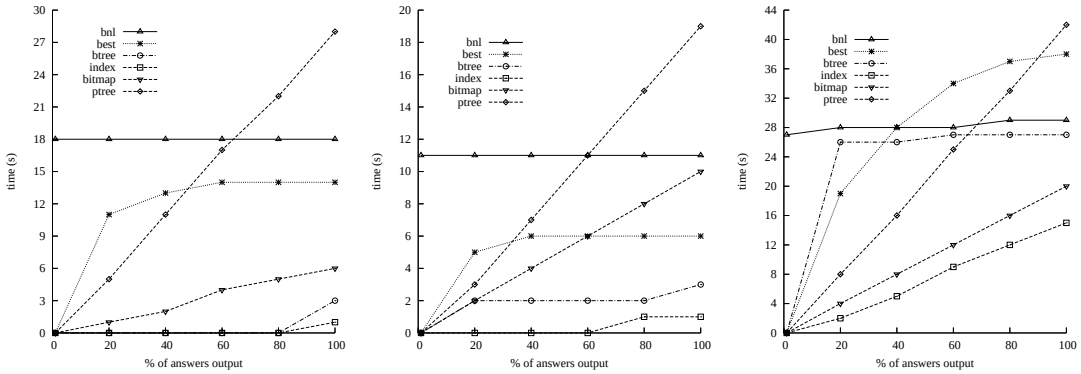
bnl's performance is the worst. Besides having to make at least one scan through the dataset, we found that it performs badly when there are preferences specified on unordered attributes as they usually result in more answer tuples. This causes more tuples to be kept in the buffer and subsequently, more comparisons are incurred, reducing its efficiency. Finally, we note that the performance of the various algorithms is generally faster for the case of 3 preferences compared to 2 preferences. Upon investigation, we found that this is because most of the 2 preference queries we extract our timings from have larger number of answers, hence, a higher runtime. However, we can expect the performance for 2 preference queries to be better in practice as the number of answers is generally smaller.

In summary, the results show that our schemes are able to produce the first few answers quickly, especially in the case of index, making them suitable for online,

interactive applications that require a fast initial response.

5.4.2 Progressiveness of the Algorithms

In this experiment, instead of examining just the time to output the first 100 results, we examine the performance of the schemes in terms of how fast *all* the answers are returned *progressively*. We ran the experiments using the 3 sets of test queries and recorded the average time taken for each algorithm to output the first tuple (close to 0%), 20%, 40%, 60%, 80% and 100% of the answers. Figure 5.16 shows our results for independent datasets.



(a) 2 preferences (b) 3 preferences (c) 4 preferences

Figure 5.16: Interval timings, independent datasets.

From the figure, several observations can be made. First, the runtimes of all algorithms increase with increasing number of preferences. It is well known in the literature that the number of answers increases with increasing number of preferences specified in the queries. Consequently, more time is required to find all the answers and higher runtimes are expected (the exception is queries with three specified preferences which we have explained why in the previous experiment).

Among all the algorithms, index is able to produce answers progressively faster than the rest. Recall that in phase two of index, we exploited the fact that preferences specified on unordered attributes usually have very few levels in their better-than graphs. Thus, we do some pre-computation prior to evaluating each query to speed up the subsequent processing. Consequently, this has made the overall

processing very efficient which we observed in several of our experiments.

For bitmap, although it is able to produce first answers fairly quickly, its performance in terms of producing answers progressively is poorer. This is due to the size of each bit-slice being large and thus causes a substantial amount of I/Os as well as processing cost to be incurred. However, notice that bitmap is less affected when the number of preferences increases (although still poorer than index). This is a direct consequence of not needing to retrieve and compare actual tuples in bitmap. Hence, its performance does not drop as significantly compared to the rest.

For ptree, its performance in terms of progressiveness is generally poorer compared to the rest. However, it is still able to produce 30% to 60% of the answers significantly faster than bnl and best which require at least one scan of the original dataset and multiple scans of subsets of the dataset. Furthermore, its progressiveness does not drop drastically when the number of preferences increases.

For btree, its performance is again worse for larger number of preferences. But, this time, it is due to the large number of *tids* that have accumulated in the first phase of the algorithm (especially when a lot of entries with incomparable values are involved). This implies that many tuples will have to be retrieved and processed by the block nested loop algorithm, causing a drop in performance.

For bnl, its performance is relatively unchanged compared to the previous experiment. In the case of 2 and 3 preferences, the number of answers of our test queries are not very large and most can fit in the window of bnl. Hence, after one scan of the original dataset, all the answers are captured in the window and output at almost the same time. For the case of 4 preferences, there are more answers and hence, more scans are required, resulting in less uniform timings.

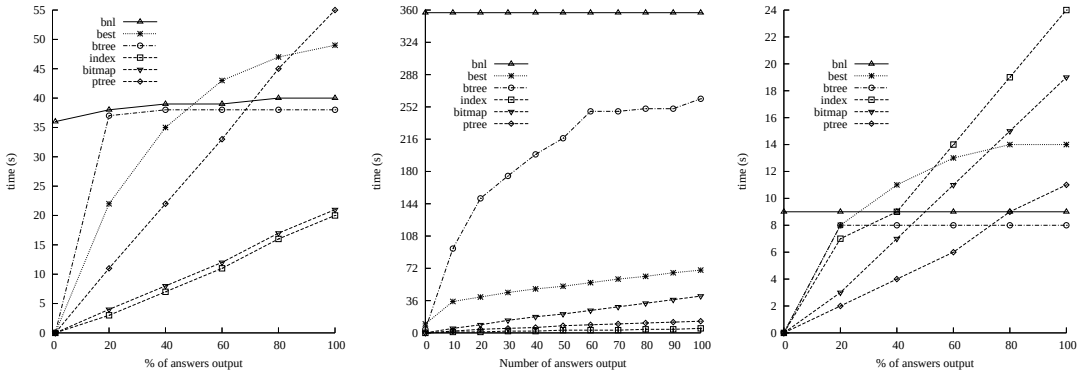
For best, we can observe that the first 20% of the answers are usually produced less progressively. This is because in the initial scans of best, there are still quite a number of non-answer tuples. Hence, more comparisons are incurred, resulting in less progressiveness in the beginning. As more non-answer tuples are eliminated, the unresolved set will get smaller and processing efficiency improves. Such a

pattern, however, is hardly noticeable for large number of preferences. This is because there are now more answer tuples and hence, the unresolved set remains large and its progressiveness does not improve as illustrated in Figure 5.16(c).

Finally, by looking at the time 100% of the results are produced, we can observe the algorithms' performance in terms of overall runtime. We can see that index outperforms the rest of the algorithms in terms of overall runtime. bitmap is good for large number of preferences while ptree is the worst in terms of overall runtime. Although bitmap and ptree are not particularly attractive in terms of overall runtimes, we believe that their progressive nature will still make them attractive options in interactive environments. In summary, the results show that our proposed schemes are able to produce answers progressively with index having the best overall performance. This makes them attractive approaches for evaluating pareto preference queries.

5.4.3 Other Experiments

We also conducted other types of experiments. For brevity, we will present only a representative set of results here (see Figure 5.17).



(a) Anti-correlated dataset (b) 1 million tuples (c) Skyline queries

Figure 5.17: Other experiments.

First, Figure 5.17(a) shows the interval timings for an anti-correlated dataset using queries consisting of 4 preferences. The experimental setup is similar to previous experiments on independent datasets. From the figure, we can observe that

the relative performance of the various algorithms remains relatively unchanged compared to using independent datasets except for higher runtimes. This is consistent with the observation in [9] where anti-correlated datasets generally result in more answers and hence, higher runtimes. It is interesting to note that bitmap's performance is now relatively close to that of index's, indicating the suitability of bitmap in answering queries that typically result in a larger number of answers.

Second, Figure 5.17(b) shows the results for returning the first 100 answers for an anti-correlated dataset consisting of *one million* tuples. The queries used consist of 3 preferences. The goal of this experiment is to test the scalability of the various algorithms. For the figure, we can see that all our proposed schemes are able to outperform the rest of the algorithms. The first answer produced remains almost instantaneous. index performs the best again. However, ptree now performs better than bitmap. This is because for a one million tuples dataset, the bit-slice becomes even larger and hence the overheads of manipulating them become significant, causing a drop in performance for bitmap. For btree, it took 4s to produce the first answer and its performance deteriorates rapidly subsequently. This is because for a large dataset, the initial scan will result in a larger number of index entries being retrieved and hence the subsequent block nested loop algorithm has to evaluate a larger number of tuples, resulting in a poorer performance. For best, it took 10s to produce the first answer but subsequent processing is relatively fast as the unresolved set gets smaller. Lastly, bnl's performance is the worst as more tuples result in more comparisons which has a detrimental effect on its performance.

Third, Figure 5.17(c) shows the results for progressive skyline computation on an anti-correlated dataset using queries consisting of 5 preferences. For this experiment, we use a combination of LOWEST and HIGHEST preferences and capture the interval timings. From the figure, we can see that index does not perform well for skyline queries compared to bitmap and ptree. In fact, ptree is the more progressive algorithm compared to the rest. Recall that index is particularly effective in evaluating queries which have preferences specified on unordered attributes.

The advantage is lost for skyline queries because the preferences are all specified on ordered attributes. This results in more iterations of the two phases, resulting in a drop in progressiveness. On the other hand, for ptree, it is fairly easy and efficient to determine whether any tuples covered by a node strictly dominates a candidate tuple. Since the preferences are only HIGHEST and LOWEST, only the largest and smallest value of the rangeset is used for checking respectively. This speeds up the processing substantially. The relative performance of bitmap, bnl and btree remains unchanged compared to the experiments conducted for skyline queries in chapter 3. It is important to note that bitmap's performance is relatively poorer compared to those experiments because we do not apply the 'early skyline' heuristic (described in [9]) here. Moreover, the absence of preferences on unordered attributes, which is an important cause of slowdown in bnl and btree, results in a better performance for both algorithms. However, the absence of such preferences has no significant impact on best. Hence, its overall progressiveness for skyline queries remains relatively similar to previous experiments.

Finally, Table 5.2 shows the construction cost of our proposed schemes as well as their space consumption for an independent dataset. We can see that the cost

	Build time (s)	Index size (MB)
index	5	12
ptree	17	34
bitmap	54	92

Table 5.2: Construction costs.

and storage required for bitmap is very much higher than the rest. This is the reason why the Bitmap scheme can only operate in more restrictive environments such as data warehouses. On the other hand, ptree is more space efficient and thus suitable for dynamic databases, although less efficient than bitmap. However, index is clearly the best here and is suitable for most environments.

In summary, this set of experimental results show that our algorithms are not only applicable for different data distributions, but are scalable and general enough to address even skyline queries.

It is important to note that while index appears to be the clear winner in most cases, we must not neglect the fact that it does not perform well for skyline queries. Hence, in an environment where a large percentage of pareto queries are skyline queries, index might not be the best option. Instead, bitmap and ptree could be used. However, while bitmap is, in general, faster than ptree, it is costly in terms of storage and is most suitable for static databases only. In such a scenario, if space is an important issue or the databases are dynamic, then ptree could be an attractive option if some loss in efficiency can be tolerated.

5.5 Summary

In this chapter, we have presented three approaches that can evaluate a broad range of pareto preference queries efficiently. Besides providing a fast initial response time, they can also return answers progressively to conserve computational resources. The first exploits a bitmap structure while the second adopts a structure similar to the R-tree. The third approach relies only the existence of single-dimensional indexes such as the B^+-tree index which is commonly available in most commercial DBMS. We also conducted an extensive performance study to evaluate the effectiveness of our approaches against existing techniques. Our results show that the third approach is the most effective in terms of initial response time and progressiveness in most cases.

CHAPTER 6

Evaluation of Numerical Preference Queries with Linear Scoring Functions

The previous three chapters have dealt explicitly with *qualitative* preference queries which specify preferences between tuples directly using binary preference relations. In this chapter, we consider *quantitative* preference queries which specify preferences indirectly using scoring functions. The scoring function is used to assign a score to each tuple. The tuples can then be ranked according to scores and returned to the users as answers.

A straightforward approach to evaluate such *numerical preference queries* would be to look at each tuple, compute its score and then order the tuples by scores before returning. However, this approach is not only inefficient but the answers can only be returned upon complete evaluation of the query.

In this chapter, we focus our attention on evaluating numerical preference queries utilizing only linear scoring functions over a relational database system. We first provide some background information in section 6.1. Then, we propose our efficient partition-based framework and algorithm that can return answers progressively as the query is being evaluated (section 6.2). A progressive approach is particularly attractive for preference queries since users will typically be interested only in the top few answers. Besides providing users with short initial response time, the system can also better utilize its resources – should the user chooses to

terminate as soon as (s)he identifies the set of satisfactory answers, there is no need to expend resources to complete the evaluation of the query.

Next, we propose three index-based partitioning strategies in section 6.3. Index-based approaches are preferred as most indexes inherently partition the databases (and hence saving the cost to partition the database at runtime), and are suitable for dynamic databases. Finally, we present our extensive performance study in section 6.4 where we evaluate our proposed approaches against existing algorithms [38, 48] which have to be adapted for answering numerical preference queries. We also evaluated our work against the PREFER system [56] which is capable of answering numerical preference queries. A summary is provided in the final section of this chapter.

6.1 Preliminaries

In the preference model proposed in [65] and described in section 2.1, preferences specified using scoring functions are known as **numerical preferences**. Consider a relation R having a set of d attributes. Each attribute A_j of R for $1 \leq j \leq d$ has an associated domain of values, $dom(A_j)$. Recall that in the framework, a numerical preference P on all d attributes of R is defined as:

$$P := rank_F(SCORE(A_1, f_1), \ldots, SCORE(A_d, f_d))$$

where f_j is a scoring function on attribute A_j, SCORE(A_j, f_j) is the SCORE base preference and F: $\mathbb{R} \times \ldots \times \mathbb{R} \rightarrow \mathbb{R}$ is the combining function. If '$<$' denotes the familiar 'less than' order on $\mathbb{R}$, then, for any two records $x = (x_1, \ldots, x_d)$ and $y = (y_1, \ldots, y_d) \in dom(A_1) \times \ldots\ dom(A_d)$, y is ranked higher than x iff $F(f_1(x_1), \ldots, f_d(x_d)) < F(f_1(y_1), \ldots, f_d(y_d))$.

For ease of reference, we shall refer to the attributes of a relation as dimensions, i.e., each record with d attributes can be seen as a point in the d-dimensional space. WLOG, we shall assume that the user specifies his (her) preferences on all d

dimensions. We discuss how to handle queries specified for fewer than d dimensions in the next section. We also assume that each dimension is normalized to a common range e.g. [0,1].

We adopt the linear scoring function for our work as it is commonly used for answering preference queries. Let $(w_1, \ldots, w_d)$ be the preference vector that represents the weights (importance) of dimensions $A_1, \ldots, A_d$ respectively where $\sum_{j=1}^{d} w_j = 1$ and $0 \leq w_j \leq 1$. For example, a tourist who is looking for budget hotel near the city may place more importance on rates and give a weight of 70% to rates, and 30% to distance. Let $x = (x_1, \ldots, x_d)$ be a record in the relation. Then, SCORE(A_j, f_j) is given by $x_j \cdot w_j$ where x_j and w_j represent the value and weight given for dimension j. Hence, the score of x is the result of applying the combining function F: $\sum_{j=1}^{d} SCORE(A_j, f_j)$ i.e. $\sum_{j=1}^{d} x_j \cdot w_j$.

We note that values in the dimension can either be maximized or minimized. For example, for a tourist, the hotel rates and distance from city should be minimized, while the rating (e.g., 3 stars, 4 stars) should be maximized. For simplicity, we shall assume that each dimension is to be maximized.

6.2 A Generic Partition-based Framework and Algorithm

In this section, we present our proposed partition-based framework and algorithm for evaluating numerical preference queries.

6.2.1 Partition-based Framework

The partition-based framework splits a dataset into partitions. The scheme is independent of how partitions are generated, and we shall defer the discussion on partitioning strategies to the next section. Once the partitions are created, we represent each partition by two "corner" points of its bounding box. We select these points such that given any set of weights, their scores bound the possible scores of all points in their partitions. Note that the "corner" points need not physically exist. We call the scores derived from these corner points as the upper

and lower bounds of a partition and they are defined as follows:

Definition 6.1 (Upper bound of a partition). Consider a partition C containing a set of d dimensional data points. Let $|C|$ denotes the size of C and p_{ij} denotes the value of dimension j of point i where $1 \leq i \leq |C|$ and $1 \leq j \leq d$. The upper bound corner point is given by point UP, $(x_1, x_2, \ldots, x_d)$, where $x_j = max_{i=1}^{|C|} p_{ij}$. In other words, for each dimension, we pick the highest value among all the points in the partition for that dimension to form the corner point. We say that the upper bound of C is given by the score obtained from UP. Intuitively, the score of UP will provide an upper bound for the scores of the points in the partition.

Definition 6.2 (Lower bound of a partition). Following the same idea as Definition 6.1, we can create the lower bound corner point LP, $(x_1, x_2, \ldots, x_d)$, such that $x_j = min_{i=1}^{|C|} p_{ij}$ for $1 \leq i \leq |C|$ and $1 \leq j \leq d$. This time, we pick the lowest value among all the points in the partition for each dimension to form the corner point. We say that the lower bound of C is given by the score obtained from LP. Consequently, the score of LP will provide a lower bound for the scores of the points in the partition.

Example 6.1. Consider the set of 8 3-dimensional data points shown in Figure 6.1(a). Assuming that the weights are 0.3 for dimension 1, 0.4 for dimension 2 and 0.3 for dimension 3. The last column of the table shows the score for each data point based on the weights. In Figure 6.1(b), we show three partitions. Consider the first partition of Figure 6.1(b). From Definition 6.1, the upper bound corner point is given by (90, 90, 90). The first dimension of this point has value 90 because it is the maximum value among all the first dimension of each point in the partition. The values of the other two dimensions are derived in the same manner. Then, following Definition 6.2, we can get the lower bound corner point which is given by (40, 40, 50).

Before we describe the query processing algorithm, we first define two important theorems that present the intuition behind the algorithm.

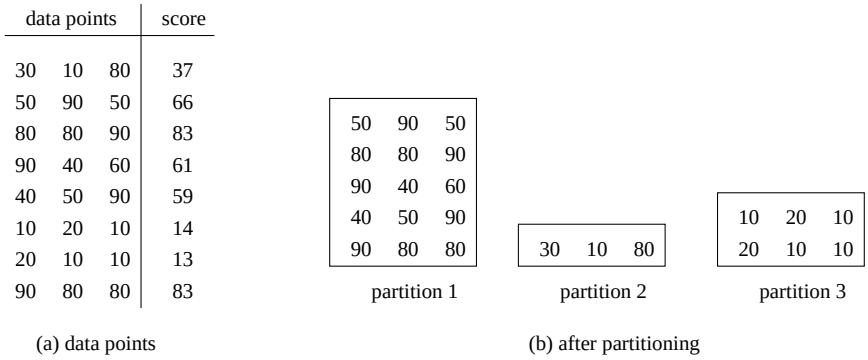


Figure 6.1: A running example.

Theorem 6.1. *Consider two partitions C_1 and C_2. Let lb_1 denote the lower bound of C_1 and ub_2, the upper bound of C_2. If $ub_2 < lb_1$, then no points in C_2 can have a score greater or equal to any points in C_1. In other words, all points in C_1 are ranked higher than those points in C_2.*

Proof (By contradiction). Assuming that $ub_2 < lb_1$ and there exists a point p in C_2 having a score $score_p$ greater or equal to some points in C_1. In other words, $score_p \geq lb_1$. Since $ub_2 < lb_1$, this also means that $score_p > ub_2$. However, since ub_2 bounds the maximum possible score for C_2, it is impossible to have such a point p in C_2. Thus, if $ub_2 < lb_1$, no points in C_2 can have a score greater than any points in C_1. □

Theorem 6.2. *Again, consider the two partitions C_1 and C_2. If $ub_2 \geq lb_1$, we cannot be sure that all points in C_2 will have a higher score than the points in C_1. However, any points in C_1 whose scores are greater than ub_2 are guaranteed to rank higher than all points in C_2.*

Proof. We first divide C_1 into 2 sub-partitions. The first sub-partition contains points having scores greater than ub_2 while the second sub-partition contains points with scores less than or equal to ub_2. Since the scores of points in the first sub-partition is always greater than ub_2, following Theorem 6.1, they are guaranteed to rank higher than points in the second sub-partition and C_2. □

The above two theorems essentially give us the key to efficiency as well as to return answers progressively. First, we can order the partitions so that we

can examine those partitions that are likely to contain higher scoring points first. Second, we can return some answers without examining all partitions. For example, consider two partitions C_1 and C_2. If they obey Theorem 6.1, then we can return points in C_1 without examining points in C_2. If the number of points in C_1 is sufficient to meet the user's need, we do not even need to evaluate partition C_2. On the other hand, if they obey Theorem 6.2, we can return points in the first sub-partition of C_1 immediately as answers.

We are now ready to look at the algorithm. Figure 6.2 shows the algorithmic description of our query processing strategy. The algorithm is highly abstracted. The input to the algorithm is a set of unordered partitions. We shall briefly discuss the routines and variables. c_i denotes partition i of the data points. Routine `order(`C`)` orders a set of partitions in non-ascending order of their upper bounds. `addTuples(`S`,` c`)` adds the data points in partition c to a partition S. `lowerBound(`S`)` and `upperBound(`S`)` find the lower and upper bounds of a partition S respectively. `score(`p`)` calculates the score of a data point based on a set of user specified weights. Finally, routine `flush(`S`)` is used to output all points in partition S in non-ascending order of the points' scores. As points are output from S, they are also removed from S at the same time.

Steps 2–3 initialize the algorithm. It first creates an empty partition S for storing data points that are potential answers. Next, it orders the partitions based on their upper bounds (step 3) in non-ascending order. We shall see in the next section how the partitions can be ordered as they are examined (when we discussed index-based partitioning strategies).

Steps 4–15 present the main part of the algorithm. Steps 4–5 add the tuples in the first partition to S and find the lower bound of S. As the points are added to S, they will be ordered in non-ascending order of their scores. Subsequently, any retrieval of points from S will also retrieve them in non-ascending order of their scores. Then, for each partition, steps 7–8 apply Theorem 6.1 by checking if the upper bound of the current partition is smaller than the lower bound of S. If it

Query Processing Algorithm
Input: A set of partitions $C = \{c_1, c_2, \ldots, c_n\}$
Output: Tuples in descending order of scores
1. // initialization
2. $S \leftarrow \emptyset$
3. $C' = \text{order}(C)$ // $C' = \{c'_1, c'_2, \ldots, c'_n\}$

4. addTuples(S, c'_1)
5. $lb \leftarrow$ lowerBound(S)
6. **foreach** c in $c'_2, c'_3, \ldots, c'_n$
7. **if** upperBound(c) $< lb$ // apply Theorem 6.1
8. flush(S)
9. **else**
10. **foreach** p in S // apply Theorem 6.2
11. **if** score(p) $>$ upperBound(c)
12. output p
13. addTuples(S, c)
14. $lb \leftarrow$ lowerBound(S)
15. flush(S)

Figure 6.2: The query processing algorithm.

is, it outputs all the points in S. Otherwise, Theorem 6.2 is applied (steps 10–12) by outputting (and removing) those points in S whose scores are greater than the upper bound of the partition under consideration. Next, the points in the current partition is added to S and the new lower bound of S calculated (steps 13–14). The whole process is repeated until all the partitions are processed.[1]

Notice that partition S is built dynamically and it is frequently accessed for applying the theorems and re-calculating the bounds and scores of points. Therefore, in our actual implementation, we maintain S as a multimap whose keys are scores of the points and the values are either pointers to the data points (if they are stored in memory) or disk offsets indicating the position of the points in a temporary file (if the points are stored on disk, as in the case when the memory buffer is full). This ensures that our implementation is highly efficient.

Example 6.2. Continuing with Example 6.1, we illustrate the query processing

[1]For ease of presentation, we have assumed that all points are to be returned according to their scores. For top-k preference queries, we can easily terminate the algorithm once the top-k answers are returned.

using Figure 6.3. Figure 6.3(a) shows the partitions after ordering them based on their upper bounds. Consider partition 1. UP is (90, 90, 90) and the upper bound is given by $90 \times 0.3 + 90 \times 0.4 + 90 \times 0.3 = 90$. LP is (40, 40, 50) and the lower bound is given by $40 \times 0.3 + 40 \times 0.4 + 50 \times 0.3 = 43$. At the start of the first iteration, records in the first partition is added to partition S. This is illustrated in Figure 6.3(b). From the figure, since the upper bound of the second partition is less than the lower bound of S, by Theorem 6.1, all the records in S are output (in sorted order based on their scores). The content of the second partition are then added to partition S. In the second iteration (shown in Figure 6.3(c)), since the third partition's upper bound is again less than the upper bound of S, all records in S can be output. The whole process then repeats itself till the required number of records are output. As shown, the framework returns answers progressively.

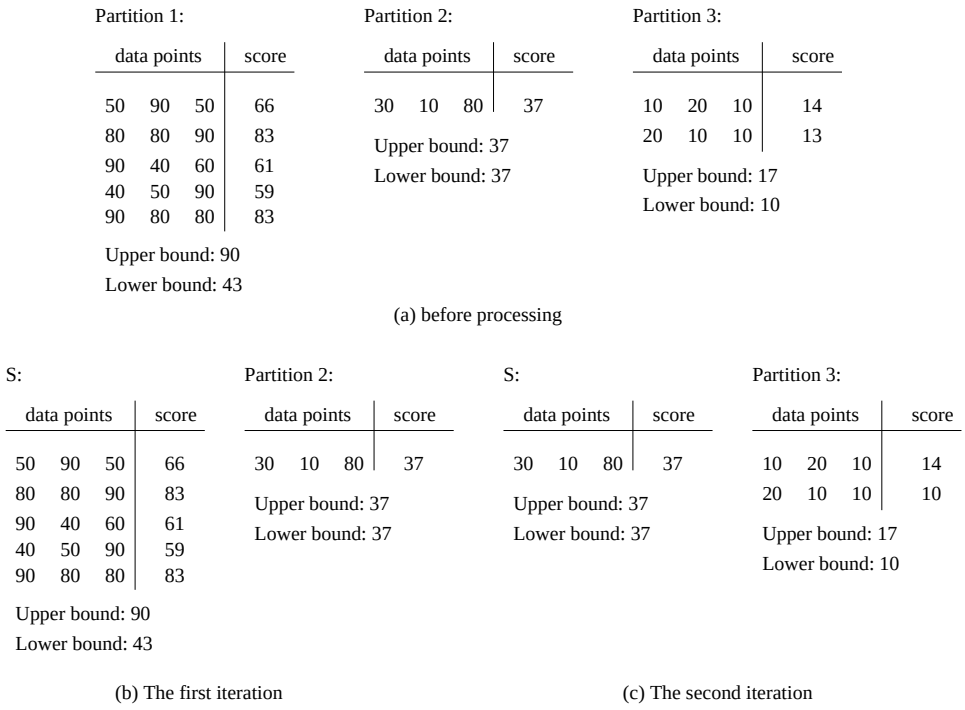


Figure 6.3: Example on how the framework works.

6.2.2 Other Issues

In the above discussions on the proposed framework, we have imposed some restrictions on the domains and number of dimensions etc. Here, we shall discuss

how the scheme can be generalized:

1. **Other domains.** We have assumed that all dimensions have the same domain. Very often, different dimensions will have different domains. For example, for a hotel relation, the domain of the `price` dimension is the set of real numbers, while the domain of the `rating` dimension is just the enumerative set containing integers 1 to 5. To handle dimensions with different domains, we only need to normalize all domains to a common one, e.g., [0,1].

2. **Number of dimensions.** We have assumed that all the d dimensions are used in the numerical preference query. For a query with fewer than d dimensions, we can set the weights of the other non-specified dimensions to 0. In this case, the proposed algorithm can be used without changes.

3. **Maximal and minimal dimensional values.** We have considered only dimensions whose values are to be maximized. For dimensions that need to be minimized, we simply transform the original value. For example, if the domain of dimension j is [0,1], and the value of a record in dimension j is k ($0 \leq k \leq 1$), we can represent k by (1 - k). The proposed algorithm remains unchanged. We note that, in most applications, it is not likely that values for a single dimension are to be maximized and minimized at the same time. For example, we would want to minimize the hotel rates and not maximize. Thus, typically, we only need to keep one set of partitions. In the unlikely case that a dimension may be used for both, we will have to keep separate partitions for our proposed scheme.

4. **Selection predicates.** The proposed scheme do not deal with any selection predicates directly. For example, it is likely that a tourist will specify additional predicates in the query, e.g., `price < 200` and/or `rating >= 3`. Clearly, these predicates can be considered while evaluating the preference query. In fact, with such constraints, the proposed scheme can be optimized to examine only relevant partitions, i.e., partitions whose points are outside

of the targeted range can be pruned away. Take the predicate `rating >= 3` as an example. As we are ordering the partitions during the initialization phase of the query evaluation, those partitions whose upper bound corner points' `rating` values < 3 can be removed from further processing. However, we note that in general, mixing selection predicates in a preference query is not a trivial issue [26, 51] and further investigations on this are required as future work.

6.3 Index-based Partitioning Strategies

In presenting the proposed partition-based framework, we have assumed that partitions are already generated. While we can scan the database and generate the partitions at runtime, this is not an efficient solution. Generating the partitions apriori (e.g., using a clustering strategy) may also not be practical especially if the database is frequently being updated. In this section, we shall look at three partitioning strategies that are supported by indexes. We shall also look at how these structures can be traversed to provide the ordering for the partitions (without explicitly sorting them apriori).

6.3.1 R-tree Based Cluster Partitioning

Our first partition-based algorithm, R-tree based partitioning, employs the R-tree [50] to index the d-dimensional points. The MBRs held at each leaf node of the R-tree essentially provide "natural" clusters for the points indexed by the R-tree. Hence, for each MBR, we can find its UP and LP as well as the corresponding upper and lower bounds. Now, one straightforward strategy is to access the MBRs and order them based on their upper bounds (see line 3 of the framework in Figure 6.2). This, however, can be computationally expensive and unnecessary (especially if the users choose to terminate as soon as (s)he gets some answers). Instead, we can traverse the tree to access these clusters in the proper order (accord-

ing to the upper bounds produced by the MBRs). Figure 6.4 shows the traversal algorithm.[2]

The algorithm is highly abstracted. Routine `Enqueue(Q,N,S)` inserts node N with upper bound S into queue Q. We note that the operation sorts the elements in Q and ordered them in non-ascending order of their upper bounds. Routine `Dequeue(Q)` removes the first element from queue Q, i.e., the element with the highest score. Function `isLeafNode(N)` returns true if node N is a leaf node and false otherwise. Finally, function `score(`p`)` computes the score of the point p.

R-tree Ordered Traversal

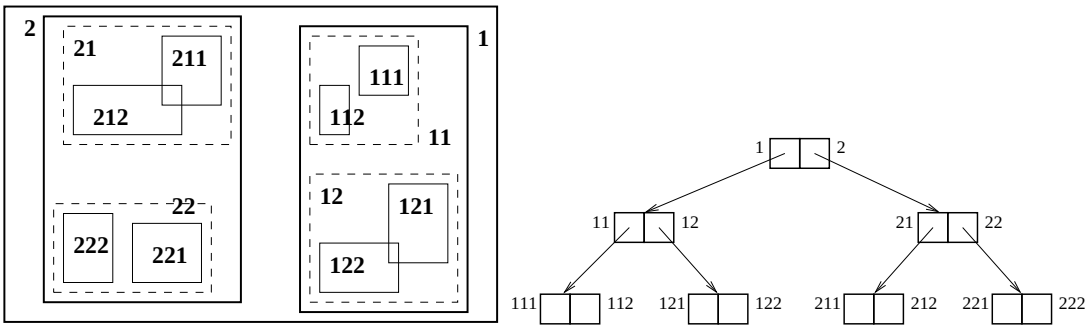
```
1.  Enqueue(Q, Root, 1)
2.  while Q ≠ ∅
3.      node ← Dequeue(Q)
4.      if isLeafNode(node)
5.          load data page pointed by node
6.      else
7.          foreach (MBR, ptr) in node
8.              S ← score(MBR's top right corner point)
9.              Enqueue(Q,MBR,S)
```

Figure 6.4: Traversing the R-tree in order.

To illustrate, consider the sample dataset (only the MBRs are shown - the points are bound within the lowest level MBRs) shown in Figure 6.5(a). The corresponding R-tree is shown in Figure 6.5(b). Initially, the root node is queued. Then both its entries, MBRs 1 and 2 are examined. Let score(MBR x) be the upper bound of MBR x. Suppose score(MBR 1) > score(MBR 2). Both MBRs are queued with MBR 1 in the front of the queue. Next, MBR 1 is dequeued, and MBRs 11 and 12 are examined. Suppose score(MBR 11) > score(MBR 2) > score(MBR 12). Then, after insertion, the order in the queue will be MBR 11, MBR 2 and MBR 12. Finally, when MBR 11 is examined, suppose its entries MBRs 111 and 112 both have scores higher than MBR 2. They are then inserted to the beginning of the queue. Eventually, MBR 111 (say with score higher than

[2]The algorithm only shows the traversal of the tree to retrieve partitions in a certain order. For evaluating preference queries, it should be integrated with the framework in Figure 6.2.

MBR 112) will be dequeued, and its points examined (as in the framework in Figure 6.2). This process is repeated as the R-tree is traversed for more partitions to be accessed.



(a) Data space (b) R-tree

Figure 6.5: An example for R-tree traversal.

6.3.2 Quad-tree Based Grid Partitioning

The main problem with the R-tree based partitioning technique is that it may result in partitions that overlap. This will mean that more partitions have to be examined before answers can be returned. An alternative approach is to adopt a hierarchical grid partitioning strategy. We begin with a grid cell that contains all points. For each grid cell that contains more than a certain number of points, we further partition it into four disjoint sub-grid cells. This process is recursively applied until a grid cell is small enough (in terms of the number of data points in it). Grid cells that do not contain any points are removed. The grid based scheme can be easily supported by a Quad-tree [94]. An example of the partitioning strategy is illustrated in Figure 6.6 using a 2-dimensional dataset. There are 7 data points and we assume that each grid cell cannot contain more than 2 data points. Since the initial number of data points exceeds the limit allowed (Figure 6.6(a)), the original grid cell is sub-divided into 4 sub-grid cells (Figure 6.6(b)). Next, since grid cell 11 exceeds the limit allowed, it is further divided into 4 sub-grid cells (Figure 6.6(c)). Finally, as grid cell 114 is empty, it is removed and a Quad-tree is built using the remaining 6 grid cells. The traversal algorithm for the Quad-tree to access grid

cells that are in non-ascending order of the upper bounds is exactly the same as that of the R-tree based scheme.

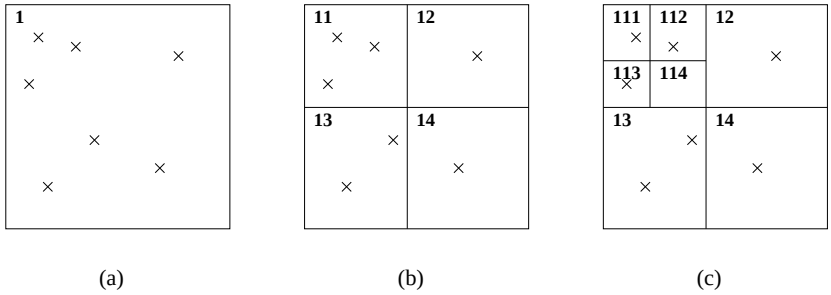


Figure 6.6: Illustration of hierarchical grid partitioning.

6.3.3 B-tree Based Edge Partitioning

In our final partitioning strategy, we partition the data points based on their "edges". An "edge" of a data point refers to the maximum or minimum value among all dimensions of the point. Since we assume that values in each dimension should be maximized, we partition points based on the maximum value in any of their dimensions. In other words, let $(x_1, x_2, \ldots, x_d)$ and $(y_1, y_2, \ldots, y_d)$ be two points in a partition. Then, $\left|\max_{i=1}^{d} x_i - \max_{i=1}^{d} y_i\right| \leq \epsilon$ where ϵ indicates the maximum difference allowed between maximum values of points in the same partition. Figure 6.7 shows an example of edge partitioning where $\epsilon = 1$. Figure 6.7(a) shows a set of 5 3-dimensional points while Figure 6.7(b) shows the partitions created. Consider partition 1. Both points are in the same partition because the difference of their maximum values among all dimensions (90 and 89 respectively) is 1.

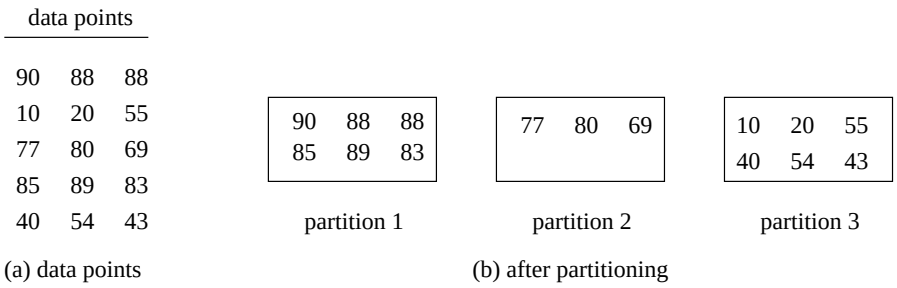


Figure 6.7: Illustration of edge partitioning.

There are two main advantages of adopting such a strategy. First, the partitions

created generally have more distinct upper and lower bounds. This can improve the efficiency of the query processing. Compare partitions 1 and 2 in Figure 6.7(b). The lower bound point of partition 1 is given by the point (85, 88, 83). Hence, the lowest score for any points in partition 1 has to be greater than 83 (given any preference vectors). On the other hand, the upper bound point of partition 2 is given by the point (77, 80, 69). Thus, the highest score for any points in this partition is less than 80. Subsequently, during query processing, Theorem 6.1 can be applied to output all points in partition 1, providing a fast initial response. Second, R-tree and Quad-tree are known to be inefficient for high-dimensional data due to the curse of dimensionality. However, since an edge is in a single dimensional space, we can adopt a single dimensional index to index the points, taking advantage of all the useful properties provided by the single dimensional access method.

For our work, we build a B^+-tree as follows. Let $(x_1, x_2, \ldots, x_d)$ be an arbitrary point. Let x_{max} be the largest value among all the d dimensions of the data point. The data point is then indexed using x_{max} as the key. Hence, the boundaries between the partitions are now delineated by key values whose difference exceed ϵ. Some bookkeeping is also done to track the boundaries so that partitions can be accessed in order of non-ascending upper bounds.

6.4 Performance Study

To evaluate the effectiveness of our proposed approaches, we conducted an extensive set of experiments to study their performance. This section reports the experimental setup and our findings.

6.4.1 Experimental Setup

All experiments are carried out on a Pentium III PC with a 866 MHz processor and 128 MB of main memory running the Linux operating system.

We implemented our proposed query processing framework together with the

various index-based partitioning strategies. We shall denote the R-tree based approach as rtree, the Quad-tree based approach as quadtree and the B-tree based approach as btree. We compare the proposed schemes against the followings:

- Threshold algorithm (denoted ta). We adapted the threshold algorithm[3] in [38] as follows. Assume a d-dimensional dataset with an index for *each* dimension. ta begins by scanning all the indexes simultaneously i.e. the first index entry of each index is accessed first, followed by the second index entry of each index and so on). As an index entry for a record r is retrieved, other index entries of r are also retrieved from the other indexes (which may involve doing random accesses to disk) and r's score is computed. The index entry of r and its associated score is then kept in memory. At the same time, ta keeps track of the last entry seen in each index and computes a *threshold value* from these entries. Let x_i, where $1 \leq i \leq d$, denotes the last indexed value seen for index i and F, the combining function. Then, the threshold value is given by $F(x_1, x_2, \ldots, x_d)$. Whenever a seen index entry's score is greater than the threshold value, its data record is retrieved and output. It has been proved in [38] that the output will be in descending order of scores.

- Quick-Combine (denoted qc). In [48], the authors proposed a basic version which is similar to ta and a full version of the algorithm which we adapted as follows. Again, assume a d-dimensional dataset with an index for *each* dimension. Initially, qc retrieves the first m entries from each index and computes an *indicator value*, $\triangle_i$, for index i. If x_i is the last index entry retrieved and y_i is the m^{th} entry that is retrieved prior to x_i in index i, then $\triangle_i$ is given by $w_i \cdot (y_i - x_i)$ where w_i is the weight specified for dimension i. Next, other index entries from other indexes corresponding to each of the m index entries are loaded and their scores computed. Then, similar to ta, a threshold value is computed and any seen records whose scores are greater than the threshold

[3]We note that an earlier algorithm, Fagin's algorithm (FA) [36], is not used as it is shown in [38] that the threshold algorithm is optimal in a much stronger sense than FA.

value are output. The most *favoured* index (based on $\triangle_i$) is accessed next. However, before the other index entries corresponding to this new index entry are retrieved, the threshold value is computed again and records that now have higher scores are output. Finally, the index entries are retrieved and a new $\triangle$ is computed for that index. The algorithm then picks the next index entry from the most favoured index for evaluation and this continues until the top k answers are output. We note that an improved version of Quick-Combine, proposed in [5, 49], is not adapted because it does not guarantee the exact order of the top k matches. Although it would be interesting to see a comparison using Quick-Combine for the case that all answers need to be found, we have omitted it in this dissertation to maintain consistency across the experiments in which all algorithms will produce answers that are ranked in non-ascending order of their scores.

- Enhanced sort based algorithm (denoted sort). This scheme makes a single pass over the dataset, and range-partitions the records into k partitions based on the scores obtained from the preference function. Assuming that partition i contains records whose scores are less than partition j for $i < j$, we can read partition j first, followed by partition $j-1$ and so on. As each partition is read, the records are sorted, and can be returned to users immediately.

The datasets used in all our experiments are generated in a similar way as described in [9]. Each dataset contains 100000 records, each of size 300 bytes. Each record has d dimensions of type *double* (in the range of [0.0, 1.0)) and one "bulk" attribute that is packed with garbage characters to ensure the record is 300 bytes (the "bulk" attribute is ignored during processing). Three types of datasets are generated: (1) In the **independent** datasets, attribute values of records are generated using a uniform distribution; (2) In a **correlated** dataset, records whose attribute values are good in one dimension are also good in other dimensions. For example, in a *Hotels* database, hotels that have higher ratings have higher room rates; (3) In an **anti-correlated** dataset, records whose values are good in one

dimension are bad in one or all of the other dimensions.

Each test query is generated by assigning a weight, w_i, in the range of [0.0, 1.0) to each of the d dimensions such that $\sum_{i=1}^{d} w_i = 1$. We assume the weights for the various dimensions follow a Zipf-like distribution [109]. Each weight for a query is generated as follows:

$$w_i = \frac{1}{i^{\gamma} \cdot \sum_{j=1}^{d} \frac{1}{j^{\gamma}}}$$

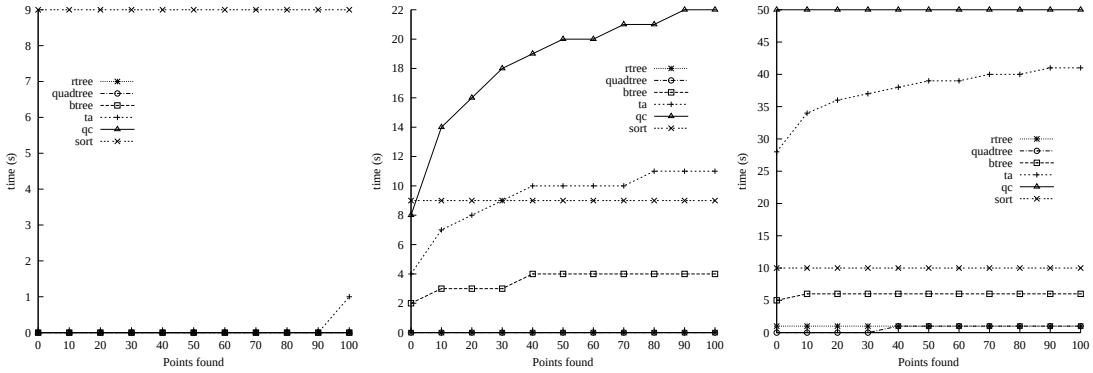
where γ is the Zipf factor of the Zipfian distribution. When $\gamma = 0$, we have the uniform distribution. When $\gamma = 1$, we have the highly nonuniform Zipf distribution. We believe that the Zipf distribution provides a good approximate model to the way weights are assigned because in practice, each user is usually focused on just a few key attributes which are given higher weights while assigning much lower weights to the rest of the attributes. In all our experiments, we set the Zipf factor to a default value of 0.8. We note that the answers to queries in all our experiments are returned ordered by non-ascending score values.

We first conducted some experimental study to tune the parameters used in the schemes. For quadtree, the maximum size of each grid cell is set to 2000 while for btree, ϵ is set to 0.001. For qc, the value of m is 100 while for sort, 10 partitions are used for range-partitioning.

6.4.2 Initial Response Time

In this experiment, we examine the performance of the various schemes in returning first few answers quickly. We recorded the time (rounded to the nearest second) each scheme took to output every 10 answers, up to 100 answers. We tested the approaches using different datasets (independent, correlated, anti-correlated) using records of dimensions 2, 5 and 8. Figures 6.8, 6.9 and 6.10 show our results for independent, correlated and anti-correlated datasets respectively.

Figure 6.8 shows the results for independent datasets. From the results, several observations can be made. First, all three of our proposed schemes produce the initial answers very quickly (instantaneous in some cases). However, btree is



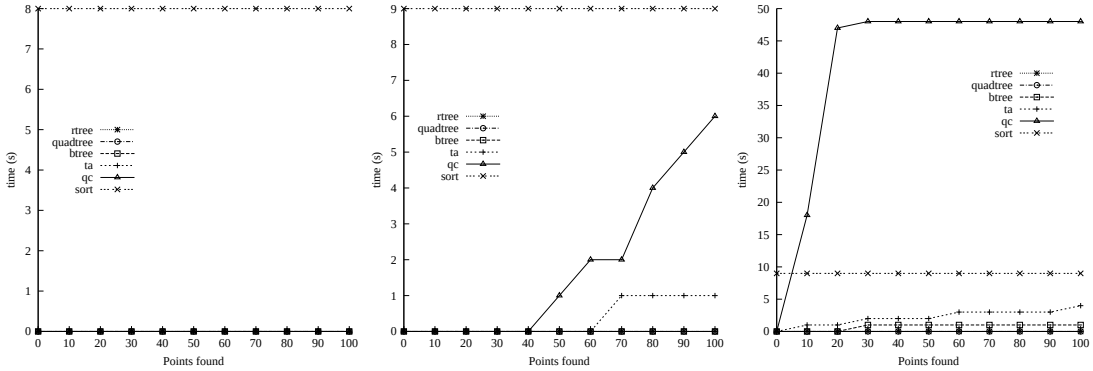
(a) dimension = 2 (b) dimension = 5 (c) dimension = 8

Figure 6.8: Timings of first 100 points for independent datasets.

generally slower for high dimensions. This is because when the number of dimensions increases, the probability of a dimension having the maximum value increases. Consequently, the upper bound of each partition in the btree will be much higher than the scores of its records. Hence, more partitions have to be examined before returning the first few answers, causing btree to perform slightly slower.

Second, sort's performance is relatively stable for all dimensions. This is because in all these cases, sorting of the first partition is sufficient to produce the answers as the partition generally contains more than 100 records. ta and qc, on the other hand, perform best when the number of dimensions is small (<5) but become worse as the number of dimensions increases. This is a direct consequence of having fewer indexes to scan when the number of dimensions is small, thereby enabling it to produce first answers faster. We also observed that qc generally produces first answers slower than ta. Recall that qc uses a heuristic approach to determine which index should be accessed next during query evaluation. We found that indexes whose dimensions are given higher weights have indicator values that remain relatively high even after many of their entries are accessed. Although this results in the retrieval of many potentially high scoring records, but since the threshold value is now decreasing at a slower rate, most of these seen records' scores will be below the current threshold value most of the time and cannot be output. Therefore, many index entries (and records) from those favoured indexes have to

be accessed before their indicator values drop sufficiently low for other indexes to be accessed to reduce the threshold value further. This explains why qc has a much higher initial response time.

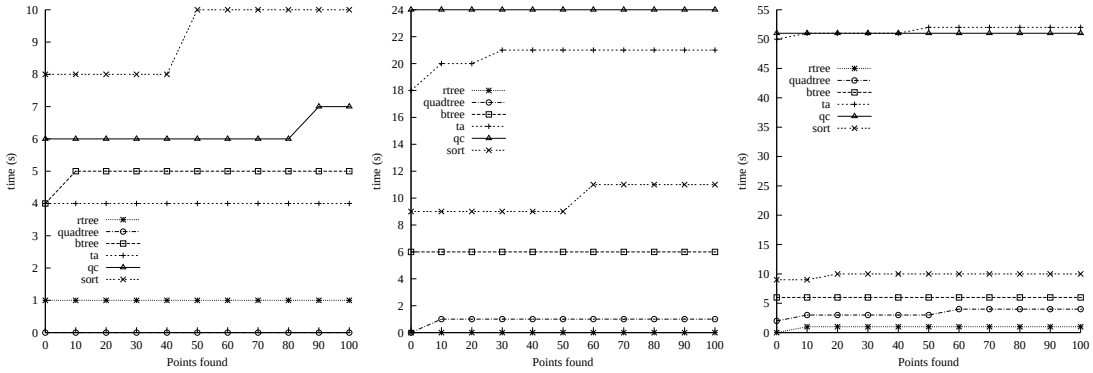


(a) dimension = 2 (b) dimension = 5 (c) dimension = 8

Figure 6.9: Timings of first 100 points for correlated datasets.

Figure 6.9 shows the results for correlated datasets. From the figure, we can see that the relative performance of the techniques remains unchanged compared to using independent datasets except that they took a shorter time. All our three proposed schemes perform well. This is because for correlated datasets, the bounds of the partitions are much closer to the scores of their records. Hence, examining the first few partitions after they are ordered is sufficient to find the first answers, resulting in a fast initial response. sort's performance only shows minor differences since the first answers can again be located by just sorting the first partition. ta's performance is also better as the index entries of most of the top scoring records are retrieved from the indexes first in correlated datasets. This reduces the initial scanning time significantly and thus its performance is better. For qc, although only 1 or 2 indexes are mainly accessed initially (see previous paragraph), some of these seen records (about 20 of them) have scores that are higher than the initial high threshold value and are thus output early, providing a faster initial response.

Figure 6.10 shows the results for anti-correlated datasets. From the figure, the relative performance of the various schemes remains unchanged compared to using the independent datasets except that they took a longer time. In anti-correlated



(a) dimension = 2 (b) dimension = 5 (c) dimension = 8

Figure 6.10: Timings of first 100 points for anti-correlated datasets.

datasets, the dimensions' values vary widely. This causes the bounds of the partitions to differ widely from the scores of the records they contain. For example, in a 8-dimensional dataset, the bounds of the first few partitions produced by quadtree are in the range of 0.8-0.9 while the records' scores in these partitions are only in the range of 0.5-0.6. Therefore, the first few partitions that are processed generally do not produce any answers but they will incur a runtime penalty. This explains why our proposed schemes generally took a slightly longer time to produce the first answers. For sort, its performance remains fairly consistent again for the same reasons discussed in previous paragraphs. For ta and qc, their performance is good for low dimensions. However, ta's performance degrades rapidly for high dimensions. This is because besides having to access more indexes for high dimensional data, a higher scanning cost is also incurred in ta as the index entries accessed initially usually do not result in high scoring records. In contrast, qc is less affected as it is designed to counter problems encountered in non-uniform datasets.

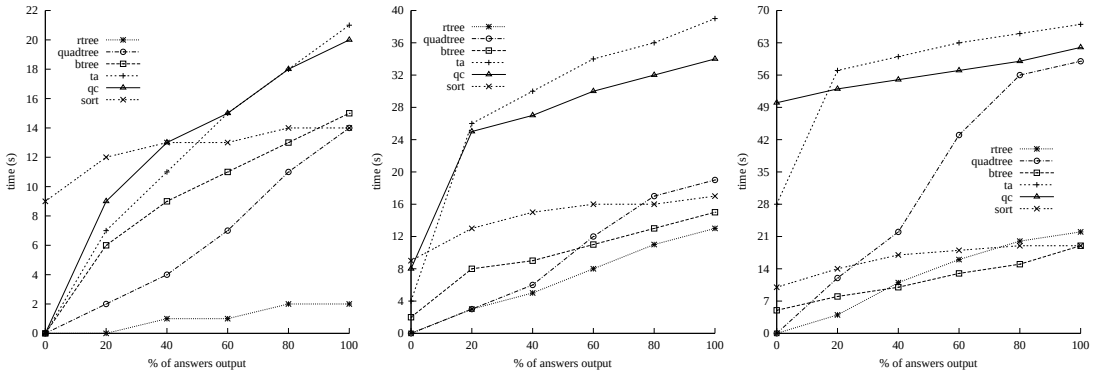
In summary, all our proposed schemes are able to perform as well as or better than the reference techniques in terms of returning first answers. Therefore, they are attractive options in interactive applications where the first few answers are crucial to answering the users' queries. In particular, the R-tree and Quad-tree schemes are the overall winners in this aspect.

It is interesting to note that the average time to produce every 10 answers (up to

100) is relatively constant for most schemes. This is because we tested the schemes using a lightly-loaded system for a single user. We believe that the deviation will be more obvious in a multi-user, heavily loaded system. However, based on the current results, we are confident that our proposed schemes will continue to perform well under such circumstances.

6.4.3 Progressiveness of the Algorithms

In this experiment, instead of examining just the time to output the first 100 results, we examine the performance of the schemes in terms of how fast *all* the answers are returned *progressively*. We tested the schemes using different datasets (independent, correlated, anti-correlated) using records of dimensions 2, 5 and 8. However, besides keeping track of the overall runtime, we also recorded the time taken for each algorithm to output the first tuple (close to 0%), 20%, 40%, 60%, 80% and 100% of the answers. All timings are rounded to the nearest second. Figures 6.11, 6.12 and 6.13 show our results for independent, correlated and anti-correlated datasets respectively.



(a) dimension = 2 (b) dimension = 5 (c) dimension = 8

Figure 6.11: Interval timings for independent datasets.

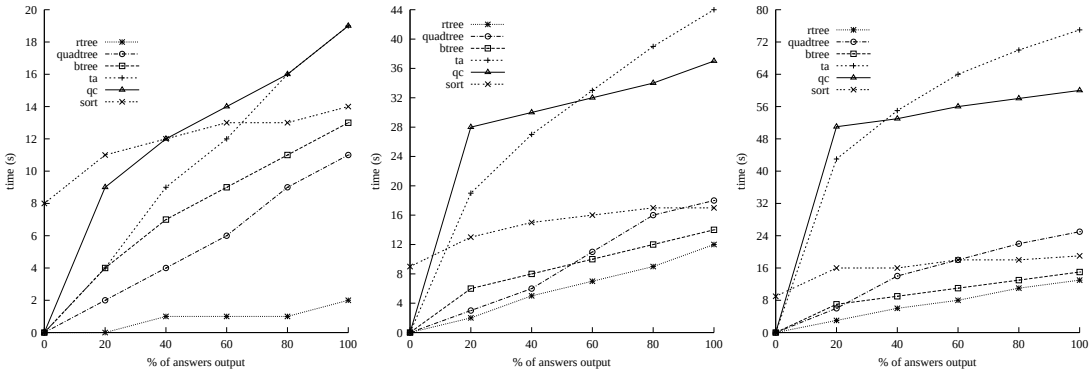
Figure 6.11 shows the results for independent datasets. From the figure, several observations can be made. First, rtree outperforms the rest of the schemes when the number of dimensions is small (<5). However, it is less progressive for high dimensions. This is because when the number of dimensions increases, there are

more overlaps in the partitions. Consequently, each partition has a higher mixture of high and low scoring records and significant overhead is required to manage those low scoring records, causing a drop in performance. quadtree also performs well for small dimensions but degrades rapidly for higher dimensions. This is because as the number of dimensions increases, the number of grid cells produced by the hierarchical partitioning decreases. Consequently, each grid cell contains more records, each with a higher mixture of high and low scoring records. Thus, more time is spent processing each grid cell and managing the low scoring records, causing the performance to decrease. For btree, it is generally slower than rtree and quadtree because the bounds of partitions from rtree and quadtree are generally *tighter* than those produced from btree. Consequently, higher processing overheads are incurred in btree. However, we note that btree is not severely affected by high dimensionality. This is because it does not make use of the spatiality of the data for its partitioning.

Second, sort's performance is relatively good compared to ta and qc and remains consistent for the various dimensions. On the other hand, ta and qc are characterized by a high initial response time for high dimensions. This is again a direct consequence of having to access more indexes for higher dimensions. It is interesting to note that ta and qc take a substantial amount of time to output the first 20% of the answers while their progressiveness are better subsequently. Recall that in ta and qc, when an index entry of a record r is accessed from an index, random accesses are made to other indexes to retrieve other index entries of r so that r's score can be computed. Moreover, not all of these records are immediately output (if their scores are below the current threshold). However, the I/O overhead incurred is significant enough to cause the first 20% of the answers to be output slower. Then, by the time 20% of the answers are output, a substantial number of records and their scores would have already been computed and hence answers can be output at a faster rate subsequently.

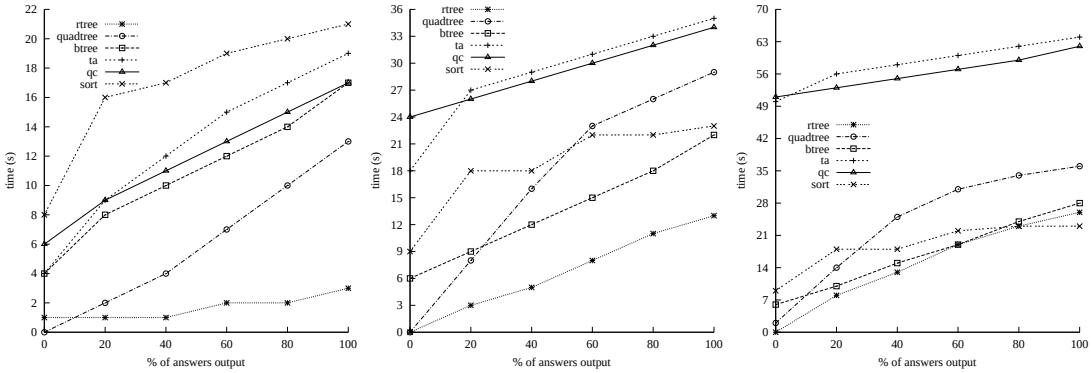
Figure 6.12 shows the results for correlated datasets. Again, the relative per-

formance of the various schemes is similar to using independent datasets except that they took a shorter time. We note that in correlated datasets, the bounds of the partitions in rtree, quadtree and btree are generally closer to the scores of their records. Hence, the three proposed schemes perform better. However, rtree is now the clear winner. sort's performance remains consistent while ta and qc are able to return initial answers faster although their progressiveness remain relatively similar to using independent datasets.



(a) dimension = 2 (b) dimension = 5 (c) dimension = 8

Figure 6.12: Interval timings for correlated datasets.



(a) dimension = 2 (b) dimension = 5 (c) dimension = 8

Figure 6.13: Interval timings for anti-correlated datasets.

Finally, Figure 6.13 shows the results for anti-correlated datasets. Again, the relative performance of the various schemes remains relatively unchanged compared to independent datasets except that they took a longer time. Recall that in anti-correlated datasets, the bounds of the partitions in rtree, quadtree and btree are very

much higher than the scores of their records. As such, there are more processing overheads and thus they take a longer time. However, all our three schemes are able to produce at least the first 20% of the answers faster than the rest, with rtree having the best overall performance. This clearly indicates the effectiveness of our schemes in answering numerical preference queries.

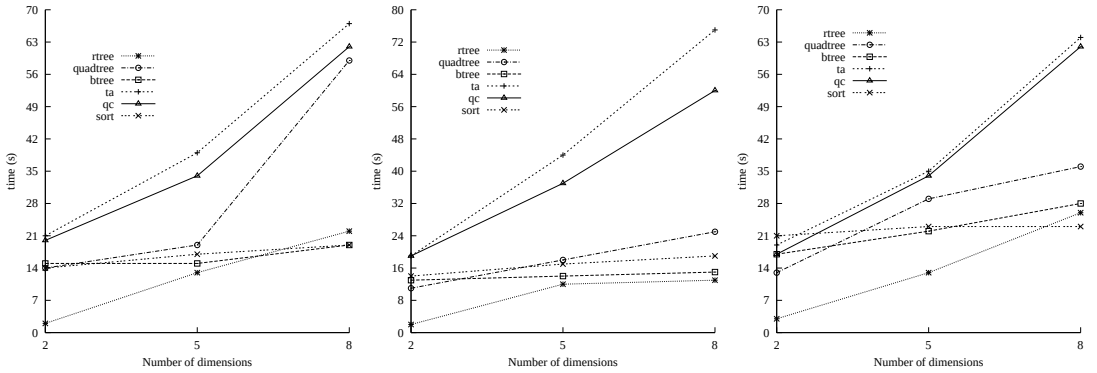
For sort, its performance is again consistent with previous experiments although we note a deviation from the norm when the number of dimensions is greater than 5. Upon investigation, we found that the number of records per partition after the initial scan is quite uneven. Hence, those larger partitions took a much longer time than usual resulting in a slight drop in performance. Finally, for ta and qc, we can see that they now take a substantial amount of time to produce even the initial set of answers. By the time the initial scan completes, a lot of records' entries would have been accessed and the scores of their respective records computed. Hence, answers are produced at a faster rate even for the first 20% of the answers as compared to using other types of datasets.

In summary, the results show that our proposed schemes besides being able to produce first answers fast, are also effective in producing answers *progressively* to the extent of returning all the results sorted by score. In particular, our R-tree scheme remains the most attractive option by outperforming the rest in most cases for different datasets of varying dimensions.

6.4.4 Comparing the Overall Runtime

In this experiment, we examine the total amount of time needed by each scheme. We tested the algorithms for each type of dataset (independent, correlated, anti-correlated) using records of dimensions 2, 5 and 8. Figure 6.14 shows the results of the experiment.

From the figure, we can see that our schemes outperform ta and qc for every dimension and for each type of dataset. We also observe that sort outperforms our proposed schemes in some cases. This is expected since our schemes are designed



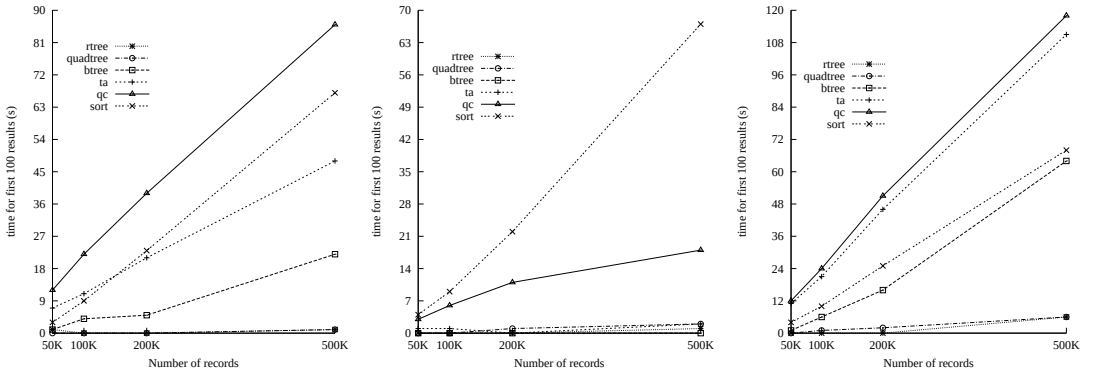
(a) Independent (b) Correlated (c) Anti-correlated

Figure 6.14: Actual runtime.

to return initial answers quickly rather than achieving a low overall runtime. The R-tree scheme, in particular, is able to outperform the rest in most cases. In conclusion, the results show that our proposed schemes not only can return answers progressively, but can outperform the other techniques in terms of overall runtime.

6.4.5 Effect of Dataset Size

In this experiment, we examine the effect of varying the size of the datasets. We recorded the time each scheme took to output the first 100 answers using 5-dimensional datasets (independent, correlated and anti-correlated) containing 50K, 100K, 200K and 500K records. Figure 6.15 shows the results of the experiment.



(a) Independent (b) Correlated (c) Anti-correlated

Figure 6.15: Varying the size of the datasets.

From the figure, we can see that all our schemes outperform the reference tech-

niques when we vary the size of each type of dataset. rtree and quadtree, in particular, have a very short initial response time compared to btree. This clearly indicates the scalability of our proposed schemes. ta and qc, on the other hand, perform badly, especially for independent and anti-correlated datasets. This is expected since our prior experiments already indicate that both schemes will not perform well for these cases. This effect is even more pronounced as we increase the size of the datasets as this incurs an even higher initial scanning cost for them, lowering their performance further. For sort, the need to scan the dataset at least once makes it unattractive for applications that require fast answers, especially in the case when the dataset is large. In summary, the results show that our proposed schemes are still able to produce first answers efficiently even though the datasets can be very large. In particular, the R-tree and Quad-tree based schemes are the most suitable for handling various type of datasets of varying sizes.

6.4.6 Evaluation Against the PREFER System

To further study the effectiveness of the proposed schemes, we also set up an experiment to evaluate them against the PREFER system [56]. PREFER is a "middleware" that sits on top of a DBMS. We adapted the source code provided by the authors to run on Microsoft Access 97 database. We also implemented the Quad-tree and B-tree based schemes as middlewares for evaluation against PREFER. For quadtree and btree, instead of building the trees, we only store the partitions, and order them at runtime (as presented in the framework in Figure 6.2). We did not compare with rtree as there are too many clusters to manage. In any case, since the R-tree based scheme has been shown to outperform quadtree and btree in most cases, the performance between PREFER and quadtree and btree would be sufficient to reflect the effectiveness of the proposed preference query processing framework. All the algorithms are programmed using Java and the data access is done by using the JDBC-ODBC bridge to connect to the Microsoft Access 97 database. The experiments are conducted on a Pentium 266 MHz machine

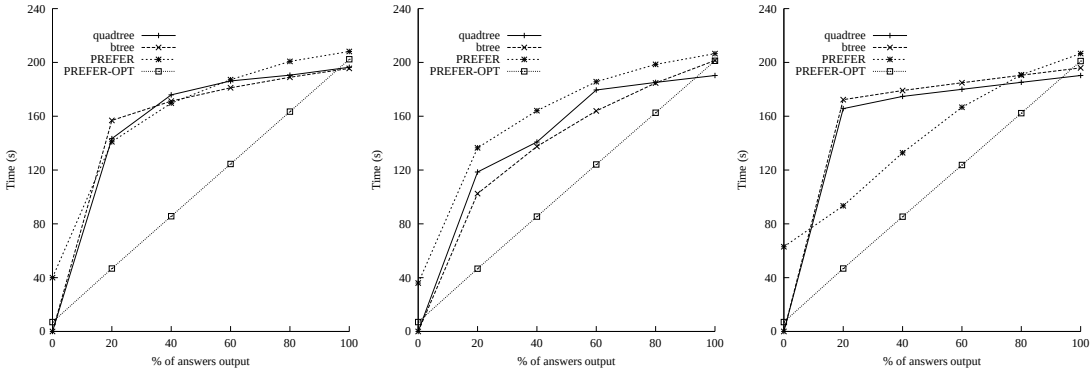
with 64 MB of RAM and 4 GB of harddisk space. In PREFER, a query to the middleware requires (1+d) SQL queries. For quadtree and btree, every partition requires one SQL query.

We note that PREFER operates with materialized views of predetermined preference vectors. As such, queries with similar preference vectors can be quickly answered by the views. However, PREFER does not guarantee correctness if the view of a preference query is not materialized, i.e., it only produces approximate answers for such queries. We used the same datasets and queries as in the earlier experiments. For PREFER, we used 30 views. This results in almost 30 times the storage space than the proposed schemes. We evaluated two versions of PREFER. In the first case, denoted PREFER-OPT, we created additional views corresponding to the preference vectors of the queries. This allows us to study PREFER at its optimal performance. The second version, denoted PREFER, answers queries based on only the 30 views that are materialized.

Figure 6.16 shows the rate at which answers are returned for the various schemes. We only present the results for $d = 5$; other dimensions show similar results. First, as expected, PREFER is worse than PREFER-OPT. This is because PREFER has to incur more time to determine the next watermark when the preference vectors of queries do not match those used in the materialized views. Second, we note that quadtree and btree perform generally better than PREFER for independent and correlated datasets; for anti-correlated dataset, PREFER is generally superior. However, in all cases, both quadtree and btree can provide the first few tuples almost instantly, while PREFER based schemes suffer a delay. While the proposed schemes only need to fetch the first partition, the PREFER based schemes require some start-up overhead to determine the best view to use, and to determine the first watermark.

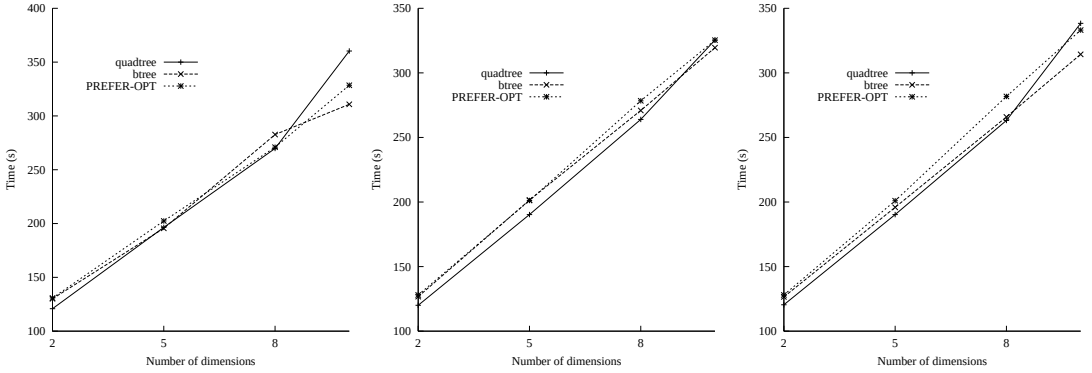
The total time results for varying dimensions are shown in Figure 6.17. We have compared quadtree and btree with PREFER-OPT (since PREFER is worse than PREFER-OPT). As shown, it is interesting to note that both quadtree and

btree perform as well as PREFER-OPT. In fact, btree is almost always slightly better than PREFER-OPT. The poor show of PREFER-OPT is attributed to the need to recompute the watermark.



(a) Independent (b) Correlated (c) Anti-correlated

Figure 6.16: Interval timings for $d = 5$.



(a) Independent (b) Correlated (c) Anti-correlated

Figure 6.17: Actual runtime.

Finally, we evaluated the proposed methods against PREFER-OPT on a constraint query that retrieves only 10% of the answer tuples. The number of dimensions used is 5, and the dataset is anti-correlated. This constraint is placed on one single dimension only (of the form $0.8 \leq d_1 \leq 0.9$ where d_1 represents the first dimension). As shown in Figure 6.18, quadtree turns out to be the best strategy here since it can pick out the relevant partitions quickly. While btree outperforms PREFER-OPT, it is inferior to quadtree as more answers are returned. This is because the "edge" is used as the partitioning criterion. As such, the sets of partitions

to be pruned is much smaller. PREFER-OPT is the worst scheme as it is unable to exploit the constraints to prune away any unwanted answers - the pruning has to be done after answers are obtained.

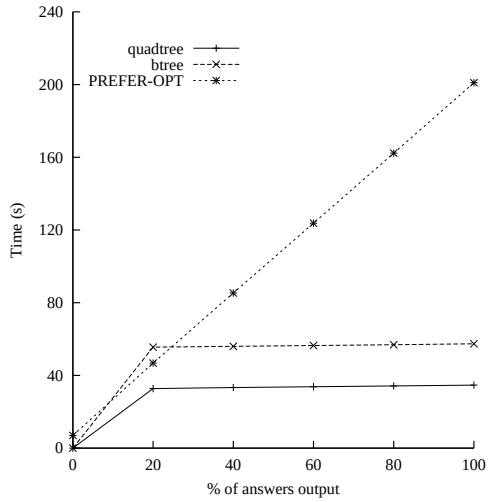


Figure 6.18: Constraint preference query.

To summarize, the proposed schemes not only can outperform PREFER, but they guarantee correctness of answers, and are capable of dealing with ad-hoc queries (with wide range of preference vectors) as well as queries involving constraints. In addition, they do not require a substantial amount of storage to maintain multiple views.

6.5 Summary

In this chapter, we have presented a partition-based framework for evaluating numerical preference queries. We have also proposed three index-based partitioning schemes: the first uses clusters obtained from the leaf nodes of a R-tree as partitions, the second adopts a hierarchical grid partitioning technique using a Quadtree, and the third partitions the data based on the maximum/minimum values of the attributes to be indexed by a B-tree. We also conducted an extensive performance study that showed the efficiency of the proposed schemes. Our results show that the proposed schemes provide short initial response time and return first few answers very quickly. Moreover, the schemes outperform existing techniques

in terms of total response time. We also evaluated our work against the PREFER system [56] which can be used directly to answer preference queries. Our evaluation against PREFER further confirms the effectiveness of our proposed schemes.

CHAPTER 7

Conclusion and Future Work

Skyline and preference query processing over relational database systems is an emerging domain of research. Traditional database queries based on *exact match* semantics frequently result in either the 'no match' effect where no answers are found or the 'flooding' effect where too many answers (mostly irrelevant) are returned. Such effects are clearly undesirable and is a source of frustration for users browsing databases for interesting information.

Supporting *best match* searches in the form of skyline and preference queries outline in this thesis provides a means to overcome the aforementioned effects. While several preference frameworks for database systems have been proposed recently, efficient structures and algorithms for evaluating such queries are still lacking. In this dissertation, several novel approaches to evaluating skyline, pareto and numerical preference queries over relational database systems are proposed. All the proposed schemes share two common notable features that are clearly important in today's applications:

1. All the techniques provide a fast initial response time by returning answers as soon as they become available. This is particularly important for interactive applications where the first few answers returned are sufficient for the users to form a big picture of the available options.

2. All the techniques are progressive i.e. they do not attempt to find all the answers in one shot. Instead, partial results are computed and returned to users initially and more results computed on demand. This allows users to terminate the processing prematurely as soon as (s)he is satisfied with the partial answers, saving precious resources in computation.

All the proposed schemes are further analyzed empirically through extensive experimental studies and the results indicate that the proposed schemes are indeed promising. We shall now reiterate specific contributions of this thesis.

7.1 Contributions

In chapter 3, we have considered the progressive computation of skylines and have proposed two algorithms for evaluating such skyline queries. The first approach (Bitmap) takes advantage of the fact that bitwise operation is fast and exploits a bitmap structure to quickly identify whether a point is in the skyline. The second approach (Index) exploits a transformation mechanism and a B^+-tree index to determine skyline points in bursts. We implemented the algorithms and evaluated them against existing techniques. We also conducted several experiments involving mix annotations which is largely ignored in most existing work. Our results indicate that our Index scheme is superior in most cases while Bitmap performs well for small number of distinct values per dimension as well as in cases where the number of skyline points is large.

Like most existing work, our work on progressive skyline computation deals exclusively with totally-ordered attribute domains. Evaluating skyline queries where domains might be partially-ordered such as set-valued domains are considered in chapter 4. We propose a framework to compute such skyline queries. The basic idea is to (a) transform each partially-ordered attribute domain into two integer-domain attributes, (b) organize the transformed attributes in an existing indexing method, and compute the skyline answers via the index. We also propose three

algorithms based on the above framework. BBS$^+$ is an adaption of BBS but lacks progressiveness. SDC exploits the properties of domain mappings to avoid unnecessary dominance checking. Finally, SDC$^+$ is an optimized version of SDC. We implemented our proposed approaches and evaluated their performance against the block nested loop algorithm (and its variant). Our results show that our proposed techniques outperform the block nested loop algorithms by a wide margin (between a factor of 2 to 16), with SDC$^+$-$MinPC$ (an optimized variant of SDC$^+$) offering the best performance both in terms of response time as well as progressiveness.

We then turn our attention to evaluating pareto queries (which is more general than skyline queries) in chapter 5. Pareto queries support a wider range of base preferences and therefore allow a broader class of preferences to be specified. This problem is challenging because existing techniques are mainly batch-oriented and only a limited subset of existing skyline strategies can be extended to answer the more general pareto queries. We propose three approaches for evaluating pareto queries. The first is a non-trivial extension of our Bitmap scheme for evaluating skyline queries. However, it suffers from the same problems of high storage and maintenance costs as in the original scheme and hence is limited to static databases such as data warehouses where updates are rare and queries frequent. To deal with dynamic databases, we propose the second approach which is a tree structure similar to the R-tree. By sacrificing some efficiency, it provides a more space efficient solution with a lower maintenance cost compared to the first approach. While these two approaches are essentially multi-dimensional indexes, the third approach is based solely on single-dimensional indexes. This makes it relatively easy to integrate into existing database systems compared to the first two approaches. However, it requires a single dimensional index to be built for each attribute that may be queried. We also conducted an extensive performance study to evaluate their performance against existing techniques. Overall, our results indicate that the last approach which is based on single-dimensional indexes is the most attractive option in terms of progressiveness and initial response time. However, it

is important to note that this approach is less effective in answering skyline queries than the other two approaches. Thus, it is not suitable for applications that need to process pareto queries consisting of a large proportion of skyline queries.

Finally, in chapter 6, we examine the evaluation of numerical preference queries based on linear scoring functions. We present an efficient partition-based framework that can return answers progressively. The basic idea is to represent each partition by two "corner" points of its bounding box: the point with the maximum score and the one with the minimum score. The partitions can then be scanned in the order given by descending values of the scores obtained from the maximum corner point of each partition. By using the maximum and minimum corner values, we can easily determine whether records that have been seen so far have scores higher than records that have not been accessed. Those (seen) records with higher values can be returned as answers immediately. We also propose three index-based partitioning strategies. Index-based approaches are preferred as most indexes inherently partition the database. The three strategies are based on the R-tree, Quad-tree and *edges* i.e. maximum/minimum value of the attributes. We implemented the proposed algorithms and evaluated their performance against existing algorithms which have to be adapted for answering numerical preference queries. We also evaluated our schemes against the PREFER system. The comparative results confirm the effectiveness of our proposed evaluation framework and algorithms.

7.2 Discussion

For this dissertation, we have introduced three different type of preference queries as well as various techniques requiring different data structures or indexes to evaluate the respective type of queries. So, is there a single universal technique or structure or index that can be used to answer all the various type of preference queries? Our answer is yes and that strategy is the block nested loop algorithm which is the most versatile not only in answering preference queries but in other type of database queries as well.

However, this versatility comes at a price. The block nested loop algorithm is generally not progressive and has poor initial response time because it needs to make at least one pass through the dataset. This has severe implications as most decision support applications today are dealing with very large dataset (making even a single pass extremely time consuming) and users of these applications are generally expecting to see results in a matter of seconds (short initial response time). In other words, while the block nested loop algorithm is versatile, it is not necessary the best approach to adopt in today's applications.

Then, do we have to implement all the structures and indexes we proposed in this dissertation in order to cater for the various type of preference queries? Our answer is no. It is impossible to have all the different data structures and indexes implemented in a single database system to answer the different type of preference queries as the cost (storage and updates) will be too costly. However, it is important to note that it is highly unlikely that every application is going to support each and every type of preferences. Trying to make an application support each and every type of preferences would make the application too complex to implement and due to the non-monotonous nature of preferences, it is difficult to achieve a high level of efficiency in answering all the various type of preference queries.

Therefore, what we have done in this dissertation is to come up with a repertoire of techniques and structures for evaluating preference queries that we can implement for an application depending on the level of support for preferences the application requires. For the rest of this section, we shall discuss how to go about selecting the appropriate techniques based on the requirements of the applications.

Firstly, most applications generally adopt either a quantitative way or a qualitative way when it comes to specifying preferences. Recall that a qualitative preference query specifies preferences between tuples directly using binary preference relations while a quantitative preference query specify preferences indirectly using scoring functions. Although scoring functions are generally less expressive, it is relatively straightforward when it comes to evaluating such queries, especially

when linear scoring functions are used. Hence, it is not surprising to see a number of applications making use of linear scoring functions as a means of supporting users' preferences.

Based on the way preferences are going to be specified in the application, we can immediately narrow down the type of techniques to use. Consider the case where the application chooses to adopt the quantitative approach. In this case, our partition-based framework and algorithm can be used. For our framework, we have proposed three index-based partitioning strategies. The first strategy utilizes the MBRs of a R-tree as "natural" partitions for the algorithm. This strategy is good if there is already an existing R-tree built on the application's data (thus saving the cost of building the index). In fact, if there is any existing multi-dimensional index on the data that can provide "natural" partitions for the algorithm, it can also be used in a similar way as the R-tree strategy.

In the absence of existing indexes that can provide "natural" partitions, we have two further choices which will involve creating "artificial" partitions. Now, if the number of attributes on which preferences can be specified is small (< 10), we can use the quad-tree based grid partitioning strategy. Otherwise, the B-tree based edge partitioning technique should be used as it avoids the curse of dimensionality that will occur if the number of attributes on which preferences can be specified on is high. We note that since both are indexes, they can also be used to support other functions e.g. range queries by the application as well.

Now, consider the case where the application decides to support qualitative preferences. We can view this support from two angles. A *simple* support will come in the form of skyline queries. Skyline queries can be furthered divided into two categories – queries that involve only totally ordered attributes and queries that involve both totally ordered and partially ordered attributes. In the first situation, we can choose to implement either our Bitmap approach or our Index approach. The Bitmap approach should be used for situations where data is more static (due to the maintenance issues that come with it) and when the skyline queries involve

a large number of dimensions. Otherwise, the Index approach should be used as our experimental study has shown that it is the more superior algorithm. In the second situation where evaluation has to take into account partially ordered attributes, then $\mathsf{SDC}^+$with $MinPC$ strategy should be used as our experimental study shows that it gives the best performance in terms of both response time as well as progressiveness.

A *complex* support for qualitative preferences will come in the form of pareto queries (which includes skyline queries as well). Without a doubt, pareto queries are the most expressive preference queries addressed so far in this dissertation but are also the most complex. We feel that if skyline queries are sufficient in supporting users' preference queries, then applications should stick to skyline queries and use the simpler evaluation strategies that come with it as they are easier to implement. However, we have proposed three approaches to handle pareto queries should the application decides to support such queries. Each approach has its strengths and weaknesses and which one to use depends very much on the situation on hand.

We propose using the B-tree based approach as the default as our experimental studies show that it is the more superior approach for most cases. However, if a large percentage of pareto queries are skyline queries, we propose that either the Bitmap-based or the R-tree based approach be used. Although the Bitmap-based approach is relatively fast, it should be restricted to databases where data are static such as data warehouses due to its weakness in handling updates. In the absence of such conditions, then the R-tree based approach should be the most appropriate approach to adopt.

In closing, we would like to re-iterate that it is almost infeasible for an application to support every type of preferences. As such, our approach to solving the preference problem by introducing a repertoire of techniques and structures in this dissertation will enable applications to pick the most appropriate strategies pertaining to their desired level of support for users' preferences. We believe that each of our approaches will in one way or another be able to find an application

where it can support users' preferences efficiently.

7.3 Future Work

There are four major directions on which we will focus our future research: First, we would like to develop cost models for skyline evaluation. Given the variety of algorithms available for evaluating skyline queries, a cost model would be invaluable as it will allow the optimizer to pick the right schemes for different situations. We plan to first tackle the problem of estimating the cardinality of skyline queries. Some preliminary work has been done in [42] and we hope to build on these work. Next, we would devise a cost model for the various algorithms and analyze how the optimizer can be extended to produce query plans that integrate the skyline operator with other relational operators.

A second direction of our research is to explore the tradeoffs of using different domain mapping functions for skyline computation with partially-ordered domains. We will also examine the evaluation of other skyline-related queries that involved such domains. We are currently exploring efficient methods to update the domain mappings and indexes when the data points are modified.

A third direction of our research is to examine space and maintenance aspects of our qualitative approaches. These aspects are not dealt with in detail in this dissertation as the focus is on evaluation strategies. First, our bitmap structure imposes a large space as well as high processing overheads for non-discrete attributes or attributes with large cardinality. We plan to examine the possibility of using compressed bitmaps in our approach. Given that our bitmap structure is augmented with additional information, this would necessitate the proposal of new compression techniques suitable for our approach. Second, we would like do a thorough analysis of the insert, delete and update operations of our preference tree structure.

The final direction of our future research is to extend our work to address more types of complex preferences in the preference framework e.g. prioritized preferences

as well as other algorithmic aspects of the framework. An immediate issue has to do with skyline and pareto queries having a widespread bad reputation of returning a large number of results. One solution to this problem is to use indifference relations as suggested in [26] but this will generally result in compositions where the preference relation is no longer of a strict partial ordering. This could potentially impact the efficiency of most algorithms as they rely on the transitivity of the preference relation. In [66], Kießling proposes the enriching of preferences with *substitutable values semantics* (SV-semantics) which will preserve the strict partial order property. We believe that our current approaches are extensible to include SV-semantics and are currently working on this.

BIBLIOGRAPHY

[1] R. Agrawal, A. Borgida, and H.V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD'89*, pages 253–262, 1989.

[2] R. Agrawal and E.L. Wimmers. A framework for expressing and combining preferences. In *SIGMOD'00*, pages 297–306, 2000.

[3] W. Balke and U. Güntzer. Multi-objective query processing for database systems. In *VLDB'04*, pages 936–947, 2004.

[4] W. Balke and U. Güntzer. Supporting skyline queries on categorical data in web information systems. In *IMSA'04*, pages 42–47, 2004.

[5] W. Balke, U. Güntzer, and W. Kießling. On real-time top k querying for mobile services. In *CoopIS/DOA/ODBASE*, pages 125–143, 2002.

[6] W. Balke, U. Güntzer, and X. Zheng. Efficient distributed skylining for web information systems. In *EDBT'04*, pages 256–273, 2004.

[7] S. Berchtold, C. Böhm, D.A. Keim, and H. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *PODS'97*, pages 78–89, 1997.

[8] C. Böhm and H. Kriegel. Determining the convex hull in large multidimensional databases. In *DaWaK'01*, pages 294–306, 2001.

[9] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE'01*, pages 421–430, 2001.

[10] C. Boutilier, R.I. Brafman, H.H. Hoos, and D. Poole. Reasoning with conditional ceteris paribus preference statements. In *UAI'99*, pages 71–80, 1999.

[11] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *TODS*, 27(2):153–187, 2002.

[12] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE'02*, pages 369–380, 2002.

[13] M.J. Carey and D. Kossmann. On saying "enough already!" in sql. In *SIGMOD'97*, pages 219–230, 1997.

[14] M.J. Carey and D. Kossmann. Processing top n and bottom n queries. *IEEE Data Engineering Bulletin*, 20(3):12–19, 1997.

[15] M.J. Carey and D. Kossmann. Reducing the braking distance of an sql query engine. In *VLDB'98*, pages 158–169, 1998.

[16] K. Chakrabarti, M. Ortega-Binderberger, S. Mehrotra, and K. Porkaew. Evaluating refined queries in top-k retrieval systems. *TKDE*, 16(2):256–270, 2004.

[17] C.Y. Chan, P.K. Eng, and K.L. Tan. Efficient processing of skyline queries with partially-ordered domains. In *ICDE'05*, 2005. accepted for publication.

[18] C.Y. Chan, P.K. Eng, and K.L. Tan. Stratified computation of skylines with partially-ordered domains. In *SIGMOD'05*, 2005. accepted for publication.

[19] C. Chang and S. Hwang. Minimal probing: Supporting expensive predicates for top-k queries. In *SIGMOD'02*, pages 346–357, 2002.

[20] Y.C. Chang, L. Bergman, V. Castelli, C.S. Li, M.L. Lo, and J. Smith. The onion technique: Indexing for linear optimization queries. In *SIGMOD'00*, pages 391–402, 2000.

[21] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. In *SIGMOD'96*, pages 91–102, 1996.

[22] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *VLDB'99*, pages 397–410, 1999.

[23] S. Chaudhuri, L. Gravano, and A. Marian. Optimizing top-k selection queries over multimedia repositories. *TKDE*, 16(8):992–1009, 2004.

[24] C. Chen and Y. Ling. A sampling-based estimator for top-k query. In *ICDE'02*, pages 617–627, 2002.

[25] J. Chomicki. Querying with intrinsic preferences. In *EDBT'02*, pages 34–51, 2002.

[26] J. Chomicki. Preference formulas in relational queries. *TODS*, 28(4):427–466, 2003.

[27] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE'03*, pages 717–816, 2003.

[28] W. Chu, H. Yang, K. Chiang, M. Minock, G. Chow, and C. Larson. Cobase: A scalable and extensible cooperative information system. *JIIS*, 6(2/3):223–259, 1996.

[29] D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[30] D. Crawford, editor. *Special issue of the Communications of the ACM on Personalization*, volume 43, August 2000.

[31] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top n queries. In *VLDB'99*, pages 411–422, 1999.

[32] P.K. Eng, B.C. Ooi, H.S. Sim, and K.L. Tan. Preference-driven query processing. In *ICDE'03*, pages 671–673, 2003.

[33] P.K. Eng, B.C. Ooi, H.S. Sim, and K.L. Tan. Efficient evaluation of numerical preference queries with linear scoring functions. Submitted for review, 2005.

[34] P.K. Eng, B.C. Ooi, and K.L. Tan. Indexing for progressive skyline computation. *DKE*, 46(2):169–201, 2003.

[35] P.K. Eng, B.C. Ooi, and K.L. Tan. Progressive algorithms for answering pareto preference queries. Submitted for review, 2005.

[36] R. Fagin. Combining fuzzy information from multiple systems. In *PODS'96*, pages 216–226, 1996.

[37] R. Fagin. Combining fuzzy information from multiple systems. *JCSS*, 58(1):83–99, 1999.

[38] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS'01*, pages 297–306, 2001.

[39] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4):614–656, 2003.

[40] R. Fagin and E.L. Wimmers. Incorporating user preferences in multimedia queries. In *ICDT'97*, pages 247–261, 1997.

[41] R. Fagin and E.L. Wimmers. A formula for incorporating weights into scoring rules. *TCS*, 239(2):309–338, 2000.

[42] P. Godfrey. Skyline cardinality for relational processing. In *FoIKS'04*, pages 78–97, 2004.

[43] P. Godfrey and W. Ning. Relational preference queries via stable skyline. Technical Report CS-2004-03, York University, 2004.

[44] K. Govindarajan, B. Jayaraman, and S. Mantha. Preference logic programming. In *ICLP'95*, pages 731–745, 1995.

[45] K. Govindarajan, B. Jayaraman, and S. Mantha. Preference queries in deductive databases. *New Generation Computing*, 19(1):57–86, 2000.

[46] L. Gravano and H. Garcia-Molina. Merging ranks from heterogeneous internet sources. In *VLDB'97*, pages 196–205, 1997.

[47] S. Guha, D. Gunopulos, N. Koudas, D. Srivastava, and M. Vlachos. Efficient approximation of optimization queries under parametric aggregation constraints. In *VLDB'03*, pages 778–789, 2003.

[48] U. Güntzer, W. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *VLDB'00*, pages 419–428, 2000.

[49] U. Güntzer, W. Balke, and W. Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC'01*, pages 622–628, 2001.

[50] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD'84*, pages 47–57, 1984.

[51] B. Hafenrichter and W. Kießling. Optimization of relational preference queries. In *ADC'05*, pages 175–184, 2005.

[52] S.O. Hansson. What is ceteris paribus preference. *Journal of Philosophical Logic*, 25(3):307–332, 1996.

[53] J.M. Hellerstein and A. Pfeffer. The rd-tree: An index structure for sets. Technical Report 1252, University of Wisconsin at Madison, 1994.

[54] A. Henrich. A distance scan algorithm for spatial access structures. In *ACM-GIS'94*, pages 136–143, 1994.

[55] G. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, 1999.

[56] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD'01*, pages 259–270, 2001.

[57] V. Hristidis and Y. Papakonstantinou. Merging results from multi-parametric ranked queries. Technical Report 174, UCSD, 2001.

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

[59] I.F. Ilyas, W.G. Aref, and A.K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB'03*, pages 754–765, 2003.

[60] I.F. Ilyas, W.G. Aref, and A.K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB Journal*, 13(3):207–221, 2004.

[61] Y. Ishikawa, H. Kitagawa, and N. Ohbo. Evaluation of signature files as set access facilities in oodbs. In *SIGMOD'93*, pages 247–256, 1993.

[62] W. Jin, J. Han, and M. Ester. Mining thick skylines over large databases. In *PKDD'04*, pages 255–266, 2004.

[63] S.J. Kaplan. Appropriate responses to inappropriate questions. In *Elements of Discourse Understanding*, pages 127–144. Cambridge University Press, 1981.

[64] S.J. Kaplan. Cooperative responses from a portable natural language query system. *AI*, 19(2):165–187, 1982.

[65] W. Kießling. Foundations of preferences in database systems. In *VLDB'02*, pages 311–322, 2002.

[66] W. Kießling. Preference queries with sv-semantics. In *COMAD'05*, pages 15–26, 2005.

[67] W. Kießling and G. Köstler. Database reasoning - a deductive framework for solving large and complex problems by means of subsumption. In *IS/KI*, pages 118–138, 1994.

[68] W. Kießling and G. Köstler. Preference sql - design, implementation, experiences. In *VLDB'02*, pages 990–1001, 2002.

[69] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB'02*, pages 275–286, 2002.

[70] G. Köstler, W. Kießling, H. Thöne, and U. Güntzer. Fixpoint iteration with subsumption in deductive databases. *JIIS*, 4(2):123–148, 1995.

[71] G. Koutrika and Y.E. Ioannidis. Personalization of queries in database systems. In *ICDE'04*, pages 597–608, 2004.

[72] H.T. Kung, F. Luccio, and F.P. Preparata. On finding the maxima of a set of vectors. *JACM*, 22(4):469–476, 1975.

[73] M. Lacroix and P. Lavency. Preferences: Putting more knowledge into queries. In *VLDB'87*, pages 217–225, 1987.

[74] M. Lacroix and A. Pirotte. Domain-oriented relational languages. In *VLDB'77*, pages 370–378, 1977.

[75] H.X. Lu, Y. Luo, and X. Lin. An optimal divide-conquer algorithm for 2d skyline queries. In *ADBIS'03*, pages 46–60, 2003.

[76] Y. Luo, H.X. Lu, and X. Lin. A scalable and i/o optimal skyline processing algorithm. In *WAIM'04*, pages 218–228, 2004.

[77] L.P. Mahalingam and K.S. Candan. Query optimization in the presence of top-k predicates. In *MIS'01*, pages 31–40, 2001.

[78] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *TODS*, 29(2):319–362, 2004.

[79] J. Matousek. Computing dominances in E^n. *Information Processing Letters*, 38(5):277–278, 1991.

[80] J. Minker. An overview of cooperative answering in databases. In *FQAS'98*, pages 283–285, 1998.

[81] A. Natsev, Y. Chang, J.R. Smith, C. Li, and J.S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB'01*, pages 281–290, 2001.

[82] S.N. Nepal and M.V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE'99*, pages 22–29, 1999.

[83] P.E. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD'97*, pages 38–49, 1997.

[84] B.C. Ooi, K.L. Tan, C. Yu, and S. Bressan. Indexing the edge: a simple and yet efficient approach to high-dimensional indexing. In *PODS'00*, pages 166–174, 2000.

[85] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD'03*, pages 467–478, 2003.

[86] D. Papadias, Y. Tao, F. Greg, and B. Seeger. Progressive skyline computation in database systems. *TODS*, 2005. accepted for publication.

[87] C.H. Papadimitriou and M. Yannakakis. Multiobjective query optimization. In *PODS'01*, pages 1–10, 2001.

[88] E. Pöppel. A hierarchical model of temporal perception. *Journal of Trends in Cognitive Science*, 1(2):56–61, 1997.

[89] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[90] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw-Hill, 1999.

[91] C. Rhee, S. K. Dhall, and S. Lakshmivarahan. The minimum weight dominating set problem for permutation graphs is in nc. *Journal of Parallel and Distributed Computing*, 28(2):109–112, 1995.

[92] D. Rinfret, P.E. O'Neil, and E.J. O'Neil. Bit-sliced index arithmetic. In *SIGMOD'01*, 2001.

[93] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD'95*, pages 71–79, 1995.

[94] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1989.

[95] R.E. Steuer. *Multiple criteria Optimization.* Wiley, New York, 1986.

[96] I. Stojmenovic and M. Miyakawa. An optimal parallel algorithm for solving the maximal elements problem in the plane. *Parallel Computing*, 7(2):249–251, 1988.

[97] K.L. Tan, P.K. Eng, and B.C. Ooi. Efficient progressive skyline computation. In *VLDB'01*, pages 301–310, 2001.

[98] S. Tan and J. Pearl. Specification and evaluation of preferences under uncertainty. In *KR'94*, pages 530–539, 1994.

[99] R. Torlone and P. Ciaccia. Finding the best when it's a matter of preference. In *SEBD'02*, pages 347–360, 2002.

[100] R. Torlone and P. Ciaccia. Which are my preferred items? In *PReC'02*, pages 1–9, 2002.

[101] R. Torlone and P. Ciaccia. Management of user preferences in data intensive applications. In *SEBD'03*, pages 257–268, 2003.

[102] P. Tsaparas. Nearest neighbor search in multidimensional spaces. Technical Report 319-02, Department of Computer Science, University of Toronto, 1999.

[103] M.P. Wellman and J. Doyle. Preferential semantics for goals. In *AAAI'91*, pages 698–703, 1991.

[104] E.L. Wimmers, L.M. Haas, M.T. Roth, and C. Braendli. Using fagin's algorithm for merging ranked results in multimedia middleware. In *CoopIS'99*, pages 267–278, 1999.

[105] K. Wu, E.J. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *VLDB'04*, pages 24–35, 2004.

[106] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *ICDE'03*, pages 189–200, 2003.

[107] C. Yu. *High Dimensional Indexing*. PhD thesis, Department of Computer Science, National University of Singapore, July 2001.

[108] Y. Zibin and J.Y. Gil. Efficient subtyping tests with pq-encoding. In *OOPSLA'01*, pages 96–107, 2001.

[109] G.K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison Wesley, 1949.

APPENDIX A

Derivation of Bit-Slices for the Base Preferences

This appendix describes the derivation of bit-slices $BitSlice_{>P_i}(x_i, A_i)$ and $BitSlice_{\geq P_i}(x_i, A_i)$ for those base preferences not covered in section 5.1.2. Throughout, we will assume that preference P_i is specified on attribute A_i and x_i is the value for attribute A_i of a candidate tuple.

Numerical Base Preferences

BETWEEN. Let the preference P_i be BETWEEN(A_i, [low,up]). The derivation is similar to the AROUND preference. If low $\leq x_i \leq$ up, we set $BitSlice_{>P_i}(x_i, A_i)$ to zero and $BitSlice_{\geq P_i}(x_i, A_i)$ to $OrigSlice(x_i, A_i)$. Otherwise, we first compute $distance(x_i, [low, up])$. Then, we retrieve the bit-slice whose value is the smallest value $\geq up + distance(x_i, [low, up])$ that exists in the bitmap for A_i. Next, we retrieve the bit-slice whose value is the smallest value $> low - distance(x_i, [low, up])$ from the same bitmap. $BitSlice_{>P_i}(x_i, A_i)$ is the result of executing a bitwise *exclusive or* operation on both bit-slices. From Theorem 3.1, the 1s in $BitSlice_{>P_i}(x_i, A_i)$ would also represent tuples having values in the range of $[a_{iu}, a_{iv})$ for A_i. To derive $BitSlice_{\geq P_i}(x_i, A_i)$, we execute a bitwise *or* operation between $BitSlice_{>P_i}(x_i, A_i)$ and $OrigSlice(x_i, A_i)$.

We shall now show why the derived $BitSlice_{>P_i}(x_i, A_i)$ has the property that the nth bit is set to 1 iff the attribute value for A_i of the nth tuple has a shorter distance to [low,up] than x_i.

Assume that there exists a tuple y having value y_i for attribute A_i which is set to 1 in $BitSlice_{>P_i}(x_i, A_i)$ but has $distance(y_i, [low, up]) \geq distance(x_i, [low, up])$. Let a_{iv} be the smallest value $\geq up + distance(x_i, [low, up])$ and a_{iu} is the smallest value $> low - distance(x_i, [low, up])$ that exist in the bitmap for A_i. Since y is set to 1 in $BitSlice_{>P_i}(x_i, A_i)$, y_i cannot lies between low and up, and $a_{iu} \leq y_i < a_{iv}$. Thus, there are two cases to consider when $distance(y_i, [low, up]) \geq distance(x_i, [low, up])$. First, if $y_i > up$, then $y_i \geq up + distance(x_i, [low, up])$. Since a_{iv} is the smallest value $\geq up + distance(x_i, [low, up])$, $y_i \geq a_{iv}$. Second, if $y_i < low$, then $y_i \leq low - distance(x_i, [low, up])$. Since a_{iu} is the smallest value $> low - distance(x_i, [low, up])$, $y_i < a_{iu}$. Therefore, when $distance(y_i, [low, up]) \geq distance(x_i, [low, up])$, $y_i \geq a_{iv}$ or $y_i < a_{iu}$. This is a contradiction since $a_{iu} \leq y_i < a_{iv}$. Hence, when a tuple is set to 1 in $BitSlice_{>P_i}(x_i, A_i)$, its value y_i for attribute A_i must have a shorter distance to [low,up] than x_i.

LOWEST. Let the preference P_i be LOWEST. The 1s in $BitSlice_{>P_i}(x_i, A_i)$ should represent tuples having values $< x_i$ for A_i. Since the 1s in $BitSlice(x_i, A_i)$ represent tuples having values $\geq x_i$ for A_i, executing a bitwise *not* on it will result in the bit-slice whose 1s represent tuples having values $< x_i$ for A_i. The resultant bit-slice is thus $BitSlice_{>P_i}(x_i, A_i)$. On the other hand, the 1s in $BitSlice_{\geq P_i}(x_i, A_i)$ should represent tuples having values $\leq x_i$ for A_i. Since the 1s in $PreSlice(x_i, A_i)$ represent tuples having values $> x_i$ for A_i, executing a bitwise *not* on it will result in the bit-slice whose 1s represent tuples having values $\leq x_i$ for A_i. This is thus $BitSlice_{\geq P_i}(x_i, A_i)$. In the absence of $PreSlice(x_i, A_i)$, all the bits in $BitSlice_{\geq P_i}(x_i, A_i)$ are set to 1.

Non-Numerical Base Preferences

For the non-numerical base preferences, we will only describe the derivation of $BitSlice_{>P_i}(x_i, A_i)$. $BitSlice_{\geq P_i}(x_i, A_i)$ can be easily derived by executing a bitwise *or* operation between $BitSlice_{>P_i}(x_i, A_i)$ and $BitSlice(x_i, A_i)$. We will also assume that there are corresponding bit-slices for values specified in the preferences. In the case where a specified value does not have a corresponding bit-slice in the bitmap, its bit-slice is assumed to be zero.

NEG. Let P_i be NEG(A_i, $\{v_1, \ldots, v_m\}$). To derived $BitSlice_{>P_i}(x_i, A_i)$, we first check whether x_i is in the NEG-set. If it does not exist, then we can conclude that no other tuples can have a value strictly better than x_i for A_i and hence, $BitSlice_{>P_i}(x_i, A_i)$ is set to zero. However, if x_i exists in the NEG-set, then all values *not* in the NEG-set will be strictly better than x_i. Thus, by executing a bitwise *or* on the bit-slice of each value in the NEG-set followed by a bitwise *not* operation on the resultant bit-slice, we get a bit-slice whose 1s represent tuples having values that are strictly better than x_i for A_i. In other words, if $L = BitSlice(v_1, A_i) \mid \ldots \mid BitSlice(v_m, A_i)$, $BitSlice_{>P_i}(x_i, A_i)$ is derived by executing a bitwise *not* operation on L.

POS/NEG. Let P_i be POS/NEG(A_i,$\{v_1, \ldots, v_m\}$;$\{v_{m+1}, \ldots, v_{m+n}\}$). First, if x_i is in the POS-set, then no tuples can have a value strictly better than x_i for A_i. Hence, $BitSlice_{>P_i}(x_i, A_i)$ is set to zero. Second, if x_i is not in the POS-set but in the NEG-set, then those values not in the NEG-set are strictly better than x_i. Similar to the NEG preference, we derive $L = BitSlice(v_{m+1}, A_i) \mid \ldots \mid BitSlice(v_{m+n}, A_i)$ and $BitSlice_{>P_i}(x_i, A_i)$ is given by executing a bitwise *not* operation on L. Third, if x_i belongs to neither POS-set nor NEG-set, only values in the POS-set can be strictly better than x_i. Similar to the POS preference, $BitSlice_{>P_i}(x_i, A_i) = BitSlice(v_1, A_i) \mid \ldots \mid BitSlice(v_m, A_i)$.

POS/POS. Let P_i be POS/POS(A_i,$\{v_1, \ldots, v_m\}$;$\{v_{m+1}, \ldots, v_{m+n}\}$). First, if x_i is in POS1-set, then no tuples can have a value strictly better than x_i for A_i and $BitSlice_{>P_i}(x_i, A_i)$ is set to zero. Second, if x_i is in POS2-set, only values in POS1-

set can be better than x_i. Hence, $BitSlice_{>P_i}(x_i, A_i) = BitSlice(v_1, A_i) \mid \ldots \mid BitSlice(v_m, A_i)$. Third, if x_i is neither in POS1-set nor POS2-set, the only values that can be better than x_i must be from POS1-set and POS2-set. Thus, $BitSlice_{>P_i}(x_i, A_i)$ is given by $BitSlice(v_1, A_i) \mid \ldots \mid BitSlice(v_m, A_i) \mid BitSlice(v_{m+1}, A_i) \mid \ldots \mid BitSlice(v_{m+n}, A_i)$.

APPENDIX B

Strictly Dominates Semantics of the Pref-Tree Algorithm

In the Pref-Tree algorithm (Figure 5.5), the `strictlyDominates` routine uses the set s_i of an entry e to determine whether it is possible for some tuples covered by e to be strictly better than the candidate tuple x with respect to attribute A_i and preference P_i. Different heuristics are adopted for each base preference. In this appendix, we shall describe the heuristics we used. Throughout, we shall use x_i to represent the candidate tuple's value for attribute A_i and s_i the set for attribute A_i in the bounding set of an entry e. P_i is the base preference specified on A_i.

Numerical Base Preferences

Since numerical base preferences are specified on ordered attributes, the set s_i is a rangeset of the form $\{[a_1, b_1], [a_2, b_2], \ldots, [a_m, b_m]\}$ where $a_i \leq b_i$ and $b_i < a_j$ whenever $i < j$. m denotes the number of ranges in the rangeset.

AROUND. Let the preference be AROUND(A_i, z). We first determine the minimum distance, $mindist_i$, of each range in the rangeset and the desired value z for $1 \leq i \leq m$. The distance between a range $[a_i, b_i]$ and value z is 0 if $a_i \leq z \leq b_i$ or $min(abs(a_i - z), abs(b_i - z))$ otherwise. Next, we determine $p = min_{i=1}^{m} mindist_i$.

Intuitively, this is the shortest distance to z any tuples covered by e can have for attribute A_i. Next, we determine the $distance(x_i, z)$. Thus, if $p < distance(x_i, z)$, it is possible that some tuples covered by e have values for A_i that are strictly better than x_i.

BETWEEN. Let the preference be BETWEEN(A_i,[low,up]). The heuristic used is exactly the same as the AROUND preference except that distances are computed with respect to the range [low,up] instead of z.

HIGHEST, LOWEST. For the HIGHEST preference, we simply compare the largest value in the rangeset, b_m, against x_i. If $b_m > x_i$, then it is possible that some tuples covered by e have values for A_i that are strictly better than x_i. For the LOWEST preference, the heuristic is similar except that we check whether a_1, the smallest value in the rangeset, is strictly less than x_i.

Non-Numerical Base Preferences

Since non-numerical base preferences are specified on unordered attributes, the set s_i is a set signature which is a combination of the signatures of values covered by s_i.

POS. For the POS preference, we first check whether x_i is in the POS-set. If it is, then no tuples can have a value for A_i that is strictly better than x_i. Otherwise, only values in the POS-set can be strictly better than x_i. Thus, we check whether any values in the POS-set exists in s_i. If there is, it indicates that there is a possibility that some tuples covered by e have values for A_i that are strictly better than x_i. In the presence of false drops, values from the POS-set might be falsely deduced to be in s_i, resulting in the searching of additional branches that can be avoided. However, this does not affect the correctness of the algorithm.

NEG. For the NEG preference, we first check whether x_i is in the NEG-set. If it is not, then we can conclude that no tuples can have a value for A_i that is strictly better than x_i. Consider the case where x_i is in the NEG-set. We can take values that are not in the NEG-set from the domain of A_i and check whether they are in s_i. However, this is not only computationally expensive, especially for

large domains, but in the presence of false drops, a value that does not exist in the NEG-set might be falsely deduced to be in s_i when in fact, it does not. This could cause the search to exclude a branch that it should search. Hence, we take the conservative approach and assume that when x_i is in the NEG-set, there is a possibility that some tuples covered by e have values for A_i that are strictly better than x_i.

POS/NEG. There are three cases to consider. First, x_i is in the POS-set. In this case, no tuples can have a value for A_i that is strictly better than x_i. Second, x_i is in the NEG-set. Similar to the NEG preference, we take the conservative approach by assuming that some tuples covered by e have values for A_i that are strictly better than x_i. Lastly, if x_i is in neither POS-set nor NEG-set, then only values in the POS-set can be strictly better than x_i. We thus use the heuristic adopted for the POS preference in this case.

POS/POS. There are also three cases to consider. First, x_i is in the POS1-set. In this case, no tuples can have a value for A_i that is strictly better than x_i. Second, x_i is in the POS2-set. Since only values in the POS1-set can be strictly better than x_i, we adopt the same heuristics used in the POS preference. Lastly, if x_i is in neither POS1-set nor POS2-set, then only values in POS1-set and POS2-set can be strictly better than x_i. Thus, we adopt the heuristic used for the POS preference except that the POS-set now consists of values from the POS1-set and the POS2-set.